E/VPL

A System for Modelling and Enacting Software Processes

A Thesis Submitted for the Degree of Master of Science

by

Rory O'Connor B.Sc.

School of Computer Applications
Dublin City University
Ireland

August 1995

Supervisor: Professor Anthony Moynihan

DECLARATION

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science in Computer Applications is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Rory O'Connor **Date:** 28/8/95

Rory O'Connor

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my supervisors Tony Moynihan and Robert Cochran, for their help, interest and encouragement over the last two years.

I would also like to thank the Centre for Software Engineering, Dublin City University, for sponsoring this work and all the staff at the Centre for Software Engineering for making it such a pleasant place to conduct this research.

I owe a large debt of thanks to Stephen Sibbald (Land Software Engineering Centre, Department of National Defence, Canada), who was always willing to discuss my problems with process programming and share his experiences and ideas about VPL.

I would like to thank my family for all the support and encouragement they have shown me during my prolonged existence as a student and especially Kevin for proof-reading my thesis and for buying me my first computer.

Finally, I would like to acknowledge the significant input of my girlfriend Emma. On many occasions I must have bored her by talking about some problem I was having with the thesis, but she always listened and made be believe that I could do it. I would like to thank her for her patience, encouragement and faith in me.

TABLE OF CONTENTS

ABSTRACT

This research addresses the technical issues involved in specifying and mechanically supporting software development processes and is related to the view of processes as "software", i.e. as a specifiable and executable entity.

Software processes can be described using textual and graphical techniques. This allows interested parties to agree that it reflects the true process, to reason about the process and to identify potential improvements. In designing new or improved processes, an ability to simulate these processes is invaluable. Such simulations, based on the process descriptions, allow one to step through the process tasks in an interactive manner. Thus one can evaluate the effectiveness of processes, assess their behaviour and ask "what-if" questions based upon proposed modifications. Simulations with the help of quantitative data, can be run for statistical purposes, where parameters can be varied.

Process descriptions can be used as a basis for process automation, as they contain much of the information needed to build a process-centred environment. However, many currently available tools, whose origins lie in process definition, allow simulation, but do not generally support real-time execution of process descriptions.

This thesis reviews the current state-of-the-art in automated systems that enact software development processes and proposes a system called Enhanced Visual Process Language (E/VPL), which is a graphically-oriented process modelling system. A prototype system has been constructed to implement E/VPL and is evaluated to assess its potential as a process modelling system.

CHAPTER 1 - INTRODUCTION TO SOFTWARE PROCESS MODELLING

1. Introduction

In recent years, many researchers in the software engineering field have identified the software development process as a key issue in obtaining higher quality products, improved productivity, more reliable and controllable projects, and other desirable effects [Ost87] [Hum89a].

It is possible to classify current work on software development processes into two broad areas [ABGM93]. The first focuses on managerial issues and aims at improving current industrial practices by focusing on organisational issues of the software development process. The most relevant work here is being done at the Software Engineering Institute (SEI) at Carnegie Mellon, USA. [Hum89a] describes the methodology developed by SEI for software process assessment, known as the "maturity framework", where the level of maturity of the process is measured against five levels; Initial, Repeatable, Defined, Managed and Optimising. Other software process improvement programmes are emerging, such as the SPICE project, which is an ISO-sponsored standard for the assessment of key processes related to the development, maintenance and acquisition of software.

The second research area addresses the technical issues involved in specifying and mechanically supporting software processes and is related to the view of a software process as "software", i.e. as a specifiable and executable entity. The definition of process representation notations was pioneered by Osterweil [Ost87], who observed that software processes can be programmed as any other software.

Following Osterweil, many software process languages have been developed to describe, analyse and guide the software development process. The great number of

proposed notations testifies that there is still much uncertainty on the best suited paradigms. Osterweil states that;

"...there is no ideal software process description."

Starke [Sta93] notes that currently there are about 60 different languages in existence to model software processes, with no systematic procedure to evaluate the applicability of these languages to real process modelling problems.

The objective of this thesis is to review the current state-of-the-art in automated systems which support software development processes and to propose a graphically oriented process modelling system to support the flexible design and maintenance of practical process models. The ultimate aim of this research is to provide a system that would be suitable for use by small- to medium-sized companies as a support for quality and process improvement initiatives.

The organisation of this thesis is as follows; This Chapter introduces the area of software process modelling and the issues surrounding the automation of software processes. Chapter 2 presents a review of the research that has been carried out in this field, with particular regard to the formalisms developed to describe software processes and the systems implemented to enact these process descriptions. Chapter 3 examines one of these systems (VPL), appraises its suitability as a software process modelling system and assesses it weaknesses. Chapter 4 attempts improve upon the weaknesses of VPL and proposes extensions to VPL (E/VPL), including a new architecture and an enhanced enaction paradigm. Chapter 5 examines a prototype system for implementing E/VPL and details a case study using E/VPL. Finally, Chapter 6 attempts to assess E/VPL's suitability as a process modelling system and proposes a future research path for continuing this research.

1.1 Environments to Support Software Processes

There is a group of environments called computer support environments that aid in the development of computer based products, but do not directly produce anything [Chr95]. The major classes of computer support environments are;

- Configuration management systems
- Workgroup products
- Process centred environments
- Software engineering environments

Configuration management systems provide the ability to manage versions of software products and their components, but generally do not emphasise tool integration or human communications.

Workgroup products provide mechanisms for allowing documents and information to be routed in office environments with a focus on data management and human communications. They include tools such as e-mail and scheduling management programs.

Process Centred Environments (PCE) provide a strong framework in which product development can be guided. Process management is the strongest component of PCEs, however, tool integration is an essential component for the execution of real-world processes. In addition a PCE must have capabilities for communications between humans.

The goal of a Software Engineering Environment (SEE) is to provide strong support for software development. Thus it has features such as the integration of software development tools and communication between these tools. However, SEEs do not have features for the high-level description of processes.

The boundary between an SEE and a PCE can be fuzzy and depends on the degree of process support that the environment provides. While tool integration is critical to the operation of an SEE, human communication is of less significance. The characteristics that distinguish PCEs from SEEs are;

- PCEs explicit focus on process mechanisms and the resulting need for process definition and enactment languages.
- PCEs emphasis on communication between, and integration of, people and their actions, rather than on communications between, and integration of tools.

Agreed-to or de-facto tool integration standards are important for PCEs, if PCEs are to connect to the wider field of CASE. Within the CASE integration community, there are several developments that are helping in this direction. For example, the conventions introduced by CORBA and PCTE are providing an impetus to third-party developers to be "compliant".

This thesis focuses on PCEs and proposes a PCE called *Enhanced Visual Process Language* (E/VPL), based on the ideas of *Visual Process Language* (VPL) with enhancements to overcome the limitations of VPL.

1.2 Implementing a Process Centred Environment

Implementing a PCE involves much more than just addressing the technology. Indeed the success of adoption rests at least equally as heavily on personnel, organisational and cultural elements. Before we look at the in-depth technical issues, we shall address some user-orientated issues. The most important question to be asked, is "What should be automated?". If not properly designed, automated support may get in the way, not help. [Chr95] includes the following tasks for automation;

- Tasks that are well understood and stable.
- Tasks that have clear interfaces to and from other tasks.
- Tasks that are tedious may be automatable, thus allowing the developer to concentrate on the more creative aspects of the job.
- Tasks where manual involvement is error prone.
- Tasks that involve significant amounts of routine communications.
- High integrity tasks, i.e. tasks where conformance to the process must be assured.

Some additional points should be noted; first, social factors that are less tangible may also have to be considered and second, prior to gaining wide experience with process automation, it is probably appropriate to start with modest-sized processes. Also, for different reasons, developers and managers may feel threatened by the introduction of a technology such as process automation. Finally, management expectations need to recognise how long an effective implementation of process automation will take. Effective strategies must be applied in order to deal with these issues.

1.2.1 Design Issues

In this section we turn our attention to the issues surrounding the development of a PCE. There are several issues that arise from the design and implementation of a process modelling language and its extensions into execution, that must be addressed [Chr95];

- Process workahead: in practice, there are many cases when preliminary work can be performed on an activity prior to the time when all the specified entrance conditions are satisfied. Inability to allow for workahead is exhibited by VPL and most other PCEs.

- Process rollback: a PCE should account for the fact that mistakes will be made during the execution of the process and it may be necessary to revert to an earlier state of the process.
- Process backtracking: forward chaining implies that activities are performed only when their entrance conditions are met. However, it could be that an agent wishes to perform an activity, some of whose entrance conditions are not met. The agent may wish to satisfy those constraint(s) by "backtracking" through the sequence of, as yet unperformed activities in order to generate the unsatisfied entrance condition.
- Process visibility: the user interface should show the graphical process model, driven by real-time execution data and indicating project status. Colour or symbolic coding of the degree of activity completion, ownership of activities, etc., would provide instant information on, and control of, the project.
- Adapting the process "on the fly": real-time modifications to an on-going process contain significant risk, but may have to be made for a number of reasons. If the process is automated, the ability to adopt the process may be restricted.

[BMcD92] has listed the following as the basic requirements for a PCE;

- Generality: the PCE should support a range of applications and development styles and hence a range of tools.
- Flexibility: the diverse nature of PCE users (project managers, programmers, etc.) requires a flexible environment that accommodates a range of needs.
- Homogeneity: users must be able to access diverse facilities consistently through a uniform interface.
- Compatibility: the massive investment already made in code, tools and training cannot be ignored when an organisation migrate from existing development approaches to PCEs.

If a PCE is to satisfy these requirements it must provide, or make it easy to provide;

- Synergy among tools to achieve productivity.
- Visibility of the development process to achieve control.
- Unambiguous communications between tools to achieve quality.
- Consistency of interfaces to promote efficiency and to provide compatibility.

1.3 Process Modelling Basics

Before we can properly explore the realm of software process modelling, we must first define exactly what we mean by the terms Process and Process Model.
For our purposes we shall define a Process as;

> *"A set of partially ordered steps intended to reach a goal."* [FH92]

While the term process is used in many different contexts, the context for this definition is software. For software development, the goal is the production or enhancement of software products or the provision of services. Other examples are the software maintenance process, the acceptance testing process, or the process development process.

A Process Model can de defined as;

> *"An abstract representation of a process architecture, design, or definition."* [FH92]

Process models are process elements at the architectural, design or definitions level. whose abstraction captures those elements of the process which are relevant to modelling. Any representation of the process is a process model. A process model can be analysed, validated and, if enacted, can simulate the process. Process models may be used to assist in process analysis, to aid in process understanding or to predict process behaviour.

1.4 Motivation for Modelling Software Processes

Now that we have defined process modelling, we must look at the reasons for modelling software processes. In this section, we will examine the objectives behind process modelling and go on to discuss the uses and advantages of process modelling. [Hum95] describes the general motivation for defining processes as;

> *"In general you will want to use a process when your end objective is to perform some repetitive activity like writing a program, producing a report, analysing a requirement, or running a test."*

In addition to producing products, however, you should also have process objectives, such as;

- To help you to plan and track your work.
- To guide you in performing tasks.
- To help you to evaluate and improve the way you do your job.

The following endeavours to express all the main objectives and goals of software process modelling; [CKO92]

1. Facilitate human understanding and communications;
   - Represent process in form understandable by humans.
   - Enable communication about, and agreement on, software processes.
   - Formalise the process so that people can work together more effectively.
   - Provide sufficient information to allow the individual / team to perform the intended process.
   - Form a basis for training personnel on the intended process.
2. Support process improvement;
   - Re-use well-defined and effective software processes on future projects.
   - Compare alternative software processes.
   - Estimate the impact of potential changes to a software process.

- Facilitate organisational learning regarding effective software processes.
- Support the managed evolution of a process.

3. Support process management;
   - Develop a project-specific software process.
   - Reason about attributes of software creation or evolution.
   - Support development of plans for a project (forecasting).
   - Monitor, manage and co-ordinate the process.
   - Provide a basis for process measurement.
4. Automated guidance in performing process;
   - Define an effective software development environment.
   - Provide guidance, suggestions, and reference material to facilitate human performance of the intended process.
   - Retain re-usable process representations in a repository.
5. Automated execution support;
   - Automate portions of the process.
   - Support co-operative work among individuals / teams.
   - Automatically collect measurement data.
   - Enforce rules to ensure process integrity.

1.4.1 Uses of Process Models

In the following section the main uses of process modelling are categorised [KTO93] and discussed, with the inter-relationships of these categories shown schematically in Figure 1.1. Before we examine these categories, I will first define the main levels under which the utilisation of process models can be categorised, in an attempt to set the context for the usage of process models. The utilisation of process models can be categorised as follows;

1. Process documentation: using passive, and often rather informal, notation for documenting information about the process.

2. Project plan generation: slightly increasing the level of formality required for representing the model. Project plans can be generated from a generic process model by adding parameters, details and customising parts of it.
3. Automation of work-flow: comprising languages and language-based graphical editors for describing processes.
4. Process analysis and simulation: a formal process description is analysed or simulated in order to assess the "run-time" behaviour of the process.

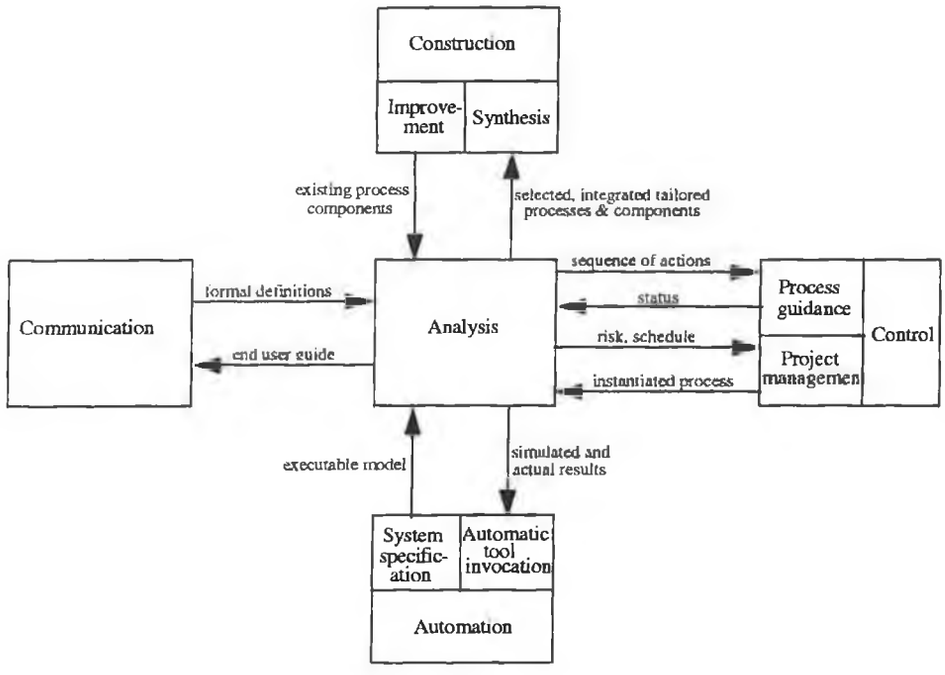

Figure 1.1 Uses of Process Modelling Formalisms

In the following paragraphs the main areas (Communication, Construction, Control, Automation) of Figure 1.1 will be explained;

Communication among process participants is very important in any project, yet situations where two or more groups on a project need to communicate, but do not, are commonplace. This can be caused by failing to realise the need for communication or each group perceiving that it is not its responsibility to initiate it. Process models are used to enable effective communication among process users, process developers, or managers in various groups involved in a project. Another use is that of transferring

process experience. It is difficult to transfer the organisation's informal process knowledge. Industrial experience shows process models can help to train new personnel by making that knowledge more formal.

Process development (or construction) is expensive, yet every organisation and even every project within an organisation, generally develops its own process from nothing more concrete than the intuition of the project managers. Humphrey [Hum90] acknowledges this situations and concludes that;

> *"...for software development to progress from a tediously unproductive craft, we must learn to build on the results of others. This starts with a defined working process."*

It is clear that the existence of explicit models of existing processes are a step in the correct direction. It is claimed [KTO93] that the problem of improving an existing process is different by degree only from the problem of synthesising a new process from existing components. Consequently, it should be expected that the capturing of an existing process in an explicit model is a useful step in its improvement.

Process control may be exerted at both the individual level (process guidance) and the organisational level (project management). The two applications are intimately related through their use of process modelling. Hubert [HFB90] points out;

> *"...the project management activity can be viewed as the instantiation mechanism for process guidance."*

Project management takes the process model, with its implied methodological constraints and uses managerial constraints such as schedules and available resources that are part of the project plan, to instantiate the process for execution. From a process perspective, process guidance takes the instantiated process and the policies and constraints it implies and controls the process at the level of the individual

participant through execution of a process model according to those policies and constraints.

Software development processes are by their nature not well understood, open-ended and non-deterministic. The partial automation of such processes involves tools that may be invoked flexibly in sequences and combinations that are not pre-specified. The area of automated process support will be examined in depth in section 1.5.

An important aspect of process modelling is that of analysis. Analysis of process models may be separated into three types, as follows;

1. Static analysis: this deals with making inferences about processes from their formal description. This may be for global optimisation or other improvement, for verification, or for validation.
2. Dynamic analysis: using simulation of the process, we can in addition to identifying flaws and problems in models and modelled processes, gain qualitative and quantitative forecasting capabilities.
3. Real-time analysis: by executing / enacting the process, rather than just a model of it, metrics and other empirical data can be collected and analysed after the fact to reveal significant properties, such as where and how the process may be improved.

By classifying the usage of process modelling and applying it to the various process modelling formalisms described in Chapter 2, it can be observed that the various authors' requirements apply to a very small set of common usage's.

1.4.2 Advantages of Process Modelling

In general the greatest advantage offered by process modelling is that it provides a vehicle for the materialisation of the process by which we develop and evolve software

[Ost87]. Through a process model, the manager of a project can communicate to workers, customers and other managers, just what steps are to be taken in order to achieve product development or evolution of goals. Programmers in particular can benefit from process models, in that reading them should indicate the way in which work is to be coordinated and the way in which each individuals contribution is to fit with others contributions.

An advantage in materialising these software process descriptions, is the ability to re-use them. At present key software process information is locked in the heads of the software managers. Others who have studied their work after they have moved on may have anecdotal views of the underlying process descriptions, but the descriptions themselves vanish with the individual who conceived them. Obviously such process knowledge is valuable and should be preserved and passed on. Materialising the process becomes critically necessary.

The preceding discussion simply emphasises that any vehicle for capturing software process knowledge is far better than no vehicle at all. There are many different forms of describing such process knowledge, including PERT charts and procedure manuals. I believe that programming process descriptions is far superior, in that it enables far more complete and rigorous description of software processes. By defining a process programming language in which data objects, data aggregates and procedural details can be captured and expressed to arbitrary levels of detail makes it possible to express software processes with greater clarity and precision than previously possible. Further, if the process descriptions are expressed in a programming language, both the act of creating the descriptions and the acts of reading and interpreting them should be comfortable for software professionals. This area of specifying processes using a process programming language will be looked at in the following section, which introduces the core ideas of automating software processes.

1.5 Automating Software Processes

In the manufacture of mechanical and electrical products, process automation has had a long time to mature. With the introduction of interchangeable parts and the assembly line in the early 1900s, automation in the form of mass production made a dramatic impact on productivity. While specific solutions in the software arena may be different, many of the underlying human and organisational issues are similar to those of automating mechanical processes. In section 1.2 we saw the major classes of automated support for software processes, in this section we will look at the philosophy behind such automated environments.

There has been a wide variety of efforts to adopt the concept of the factory to software production [Cus91]. These initiatives were motivated by a need to reduce cost and improve quality through standardising the way in which software was produced. Software factories attempted to rationalise the means of software production through a variety of techniques, using manual implementation of partial automation. Metrics data was collected in order to analyse and improve the processes and standards, reuse of software design, code and documentation was encouraged, and CASE tools were adopted.

Software process automation goes beyond what the software factories have attempted. Process automation has only recently become practical as a result of the widespread use of personal computers and workstations, and the growth in networking capability, resulting in powerful distributed computing and human communications that were not available before. It can be viewed as the next logical step in the software factory concept.

1.5.1 Process Development and Usage

Figure 1.2 proposes a process development and usage scenario [Chr95]. It provides a road map through which a process-centred environment can be implemented. This process lifecycle is analogous to the software development lifecycle.

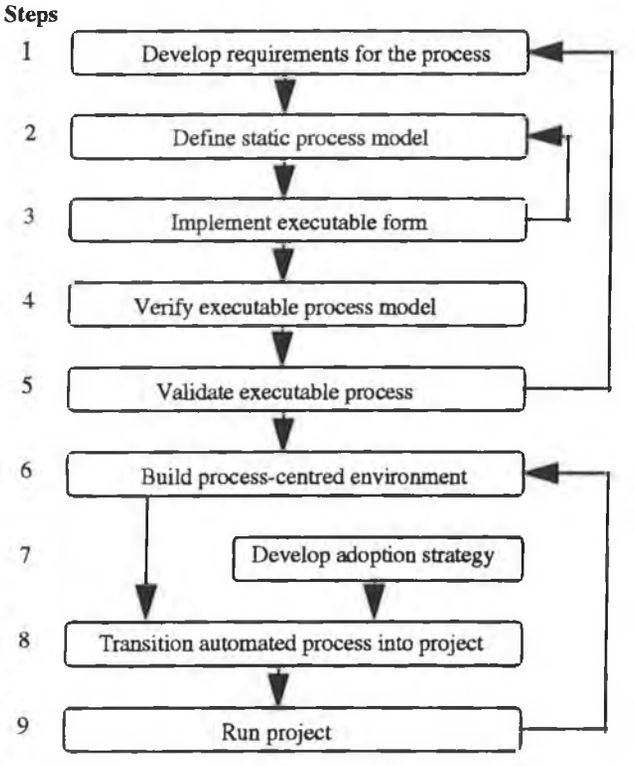

Figure 1.2 A process development and usage scenario

Prior to developing any process model, it is necessary to identify the needs of those who will work within the automated environment. From these needs a set of associated requirements can be developed (step 1). In step 2, an appropriate graphical representation of the process (the reasons for choosing a graphical representation are examined in section 2.3) that incorporates the requirements is constructed. In step 3, the graphical model is compiled into an executable form through appropriate transformations. Through this executable form, the dynamics of the process can be studied at a high level, without being encumbered, at this point, with lower level implementation details. Verifying the correctness of the model (step 4) may involve

logical (static) analysis to check deadlock and reachability and dynamic simulation to test the systems behavioural characteristics. After verification, validation of the executable model is performed (step 5).

The executable specification developed in step 3 is now used as a basis for developing the PCE. Having gained a solid understanding of the issues through this specification, development of the PCE can proceed with much greater confidence. The architecture of the PCE (step 6) may take many forms. We will investigate many of these systems in Chapter 2. The manner in which the PCE is introduced into a project is critical to its success (step 7). The automation of tasks, impersonal delegation of responsibilities and a greater degree of interaction with the computer will be issues requiring sensitive solutions. Transitioning the automated process into use is another challenge (step 8). To some extent the degree of challenge will depend on whether the automated process is being "back-fitted" into an existing manual process or being used on an entirely new process. Once an automated process is in place, process improvement may be helped significantly by the automated collection of metrics. Upon completion of the project, metrics and other data can be used to verify that the process sequence was correctly followed (step 9).

A final step in the usage (in addition to those above) is to capture demonstrably effective processes for reuse and place them in a reuse library. Executable processes leave little room for ambiguity as to what is meant. This precision is a distinct advantage in reuse over processes that have been defined but have not been developed to executability. In addition, through reuse, one automatically acquires a model that is known to work and into which process improvement lessons may have been embedded.

1.6 Summary

In this Chapter I have introduced the area of process modelling, explaining the basics of the area and motivation for modelling software development processes. We have also been introduced to the area of automated support for process models in the form of process-centred environments. In Chapter 2, we will examine several different formalisms that have been proposed to model software development processes and environments that have been developed to support these formalisms.

CHAPTER 2 - REPRESENTING AND ENACTING SOFTWARE PROCESSES

2. Introduction

In this Chapter we discuss some of the issues relating to software process representations and present some of the languages that have been proposed to model software development processes and the environments that have been developed to enact process models.

Osterweil [Ost87] proposed that the best way to achieve environments which make use of information contained in the software process, but also allow the process to be tailored to each project, is to treat the process as software. The process program should be written in a rigorously defined language. To this end researchers have developed several languages for programming and enacting the software development process.

Osterweil [Ost87] has pointed out the chicken-and-egg problem in exploring representations suited to software process modelling;

> *"In order to find out what language features we need, we need to write process programs; in order to write process programs, we need the appropriate language features."*

Another concern is which style of representation is best suited to modelling a software process. An important feature in defining process models within a software process modelling system is a graphical means of entering, browsing and editing the process model. Textual data entry and verification of a process is error-prone and it is also difficult to conceptually grasp the nature of a process model. Graphical representations should allow for the aggregation of information so that high-level abstractions may be viewed and graphical views of their components easily obtained [KTLAE92].

2.1 Example of Software Process Formalisms

In this section I shall present eight process programming languages. These vary from graphical formalisms and representations to programming languages that have been proposed in order to model software processes. This is by no means an authoritative nor exhaustive list, but is meant to act as a representative guide of the work done in the area.

2.1.1 Entity Process Models

[Hum89b] describes process models based on entities similar to those used by Jackson in the Jackson System Development (JSD) methodology [Jac83]. Here one deals with real entities and the actions performed on them. Each entity is a real object that exists and has an extended lifetime. That is, entities are things that persist rather than temporary objects that are transiently introduced within a process.

The main reasons that Entity Process Models (EPMs) can be considered useful are [Hum89b];

- EMPs deal with real objects (entities) that persist.
- Each entity is considered by itself and is viewed as having a defined sequence of states.
- State transitions result from well defined causes, although they may depend on the state of other entities as well as process events and conditions.
- As long as the relative sequential relationships of these transitions are retained within each entity stream and as long as any prerequisites and dependencies between entities are maintained, the timing within the various entity streams is not material.

The process of producing an EPM is relatively straightforward, namely;

- Identify the process entities and their states.
- Define the triggers that cause the transitions between these states.
- Complete the process model without resource constraints - a UPM.
- Impose the appropriate limitations to produce a final CPM.

EPMs based on statecharts are formal and enactable - in that we are able to run interactive, animated simulations of EPMs with STATEMATE (cf. section 2.1.6), as well as perform automated tests and analyses.

2.1.2 FUNSOFT Nets

FUNSOFT nets [EG91] [Gru93] are high level Petri nets which have been adapted to the application domain of software process modelling. FUNSOFT nets enable the expression of software specific properties by single nodes rather than (usually large) subnets in other Petri net types. The semantics of FUNSOFT nets are defined in terms of Predicate/Transition nets (Pr/T).

A FUNSOFT net contains a triple (S, T, F) which denotes a net. Elements of S are called channels and elements of T are called agencies. F denotes the set of edges. Channels are used to model object stores; agencies model activities and edges describe the relation between activities and object stores. A FUNSOFT net also contains a set of objects types O, with Bool, Integer, Real and String predefined, and a set of activation predicates P. An activation predicate can be used to ensure that an agency can fire only if tokens with certain properties are available in the preset of the agency.

In order to define the dynamic properties, like activation, firing behaviour or a reachability set, FUNSOFT nets must first be unfolded into Pr/T nets, by net morphisms. The results of this construction is used to define the dynamic behaviour of FUNSOFT nets.

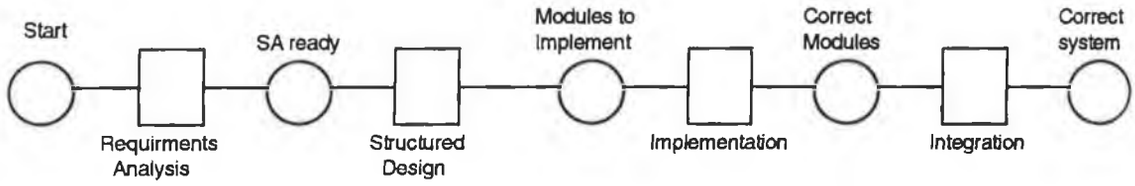

Figure 2.1 FUNSOFT representation of the Waterfall Model

Figure 2.1 depicts the Waterfall model in terms of a basic FUNSOFT net, however, for the sake of clarity, the properties and timings have been omitted. The software process management environment MELMAC (cf. section 2.2.5) supports the modelling of FUNSOFT nets, and their execution.

2.1.3 Rule Based

In [Win73] Winograd wrote of his dream of an intelligent assistant for programmers. The fundamental requirement for an intelligent assistant, he wrote, is that it understands what it does. That is, it should be based on an explicit model of the programming world. Winograd described an imaginary environment that would provide early error checking, answer questions about the program, the interactions among program parts, handle trivial programming problems and automate simple debugging tasks.

To fulfil Winograd's fundamental requirement, an intelligent assistant must understand what it does. However, there is a wide range of intelligent systems. Most software tools are moronic assistants that know what to do, but do not understand the purpose of the objects they manipulate or how their tasks fit into the development process. In other words, they know the How but do not understand the Why.

A development environment cannot "understand" why it performs an activity unless it knows [KFP88];

- The properties of the objects it manipulates.
- The systems tools and activities and the objects they manipulate.
- The preconditions under which a tool or activity can be activated.
- The results or postconditions of each activity.

An ideal environment supports the programmer in all aspects of the exploration of the problem representation and system implementation. It should be a cooperative and helpful environment in which the programmer can easily [ABCFK91];

- Inspect, modify and test programs.
- Extend and develop programs easily.
- Identify problems and deficiencies.
- Dismantle, rebuild and test programs.
- Work simultaneously on different parts of a program.
- Change the level of abstraction at which he is working.
- Get appropriate advice and assistance whenever it is needed.
- Concentrate on problem solving without having to worry about housekeeping such as file management.
- Customise the environment.

[KFP88] describes the idea of Insight, which means an intelligent assistant must be aware of the users activities and can anticipate the consequences of these activities based on an understanding of the development process and the produced software. Insight should let individual programmers become informed more quickly about the structure and relationships in the software product, to be aware of the consequences and side effects of their tasks, and to be guided in the job of making even major changes to a system and getting it back to a consistent state. It should also be able to coordinate the work of multiple programmers so they can accomplish their tasks without interfering with each other, knowing that the results of simultaneous work will be combined in a controlled way.

[KFP88] also describes the idea of opportunistic processing. That is, the automatic undertaking of simple development activities so programmers need not be bothered with them, such as determining when source code has changed, invoking the compiler, and recording errors found during the compilation (see Figure 2.2). This sort of processing is usually carried out by interpreting rules which are based on condition / action pairs. When the condition is true the action is applied. Figure 2.2 describes the rules for compiling a source file; if the file has not been compiled and all the modules are available, it is compiled and the result is a compile module or a listing of errors.

```
notcompiled(module) and
   for all components c such that in (module, component c):
      analysed(component c)
   {compile module}
compiled(module) or
errors(module);
```

Figure 2.2 Compile Rule

The limitations of rule induction are manifold, but there are two crucial problems. First is the "closed world" assumption. That is, rule induction assumes that all relevant factors are referred to in the examples - if they are not then the rules induced will not deal with all eventualities. Second, rule induction systems effectively make assumptions about test cases which do not directly correspond to these examples. Since all cases are not covered explicitly, some assumptions may prove to be ill-founded when particular examples are encountered.

In section 2.2 we will encounter several environments based on the intelligent assistant paradigm, and discuss their individual strengths and weaknesses.

2.1.4 LOTOS

LOTOS (Language Of Temporal Ordering Specification) [ISO89], has been developed for formal specification of communications systems, and has the constructs for describing concurrency, non-determinism, synchronous and asynchronous interaction, and interruption. Thus it can be considered to have enough expressive power to define the software process. Furthermore several simulators for executing LOTOS descriptions and supporting tools for verification are now being studied [Eij89]. These tools can be used to validate or enact process descriptions.

[SKS91] describes an approach to process modelling using LOTOS. It consists of two parts; one for tasks which are performed in a software process and another for resources which perform the tasks. The method for constructing a LOTOS description of a process is as follows;

1. Method for Task Part
   - Identify Tasks, Products, and Product flows: Identify the tasks and the products in a software process, and the inputs and outputs to those products. These relationships are described with a products flow diagram.
   - Identify Task Behaviour: The identified tasks may be performed in sequence, parallel or alternatively. These timings are identified and described with a timing chart.
2. Method for Resource Part
   - Identify Participants, their Relationships, and Assignment to Tasks: The participants and their relationships are represented with a entity-relationship diagram. Then an assignment of tasks to participants is made and expressed using a diagram.
   - Identify Interactions among the Participants: These are depicted in a graphical representation.
3. LOTOS descriptions are composed systematically from the intermediate products.

Figure 2.3 shows the two parts of the model; the task part for the functional aspect and the resource part for the structural aspect. The task part is modelled representing product flows generated in a process. The resource part consists of a set of participants and interaction media where participants are objects that perform tasks using the interaction media. From these intermediate products the LOTOS descriptions are produced.

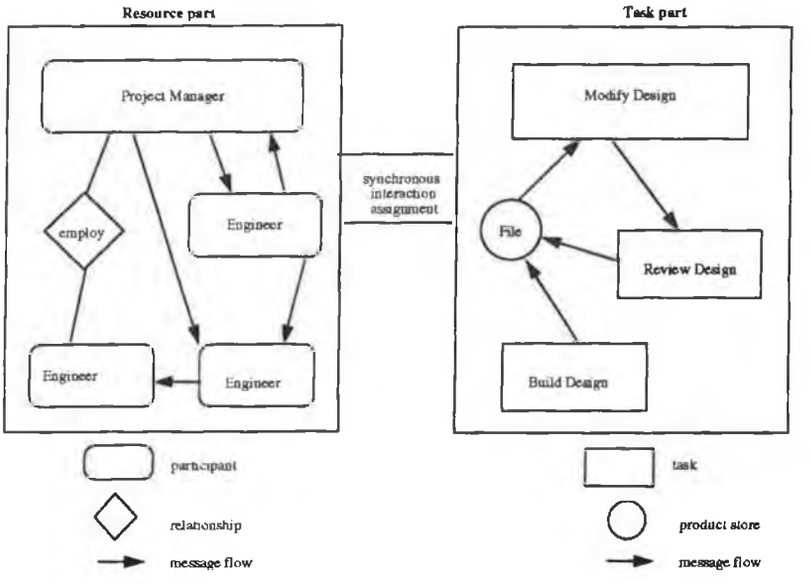

Figure 2.3 Model for resource and task part

2.1.5 PRONET

ProNet [Chr93] is a graphical process modelling language which combines ideas from object-oriented, behavioural and data-flow techniques. The ProNet model has similarities to an Entity-Relationship model in that it defines entities which are connected by relationships. A central concept in the ProNet language is the class activity. The notion of control (and hence behavioural modelling) can be embedded in the diagram through a condition class. There is also a small set of predefined relationships which link the entities. Because of the central importance of activities in the language, ProNet diagrams also have some characteristics of data-flow diagrams.

ProNet also has some characteristics similar to Petri nets, in that ProNet's notation of activities corresponds in some ways to the Petri net notation of transitions, while the other entities in ProNet correspond to the Petri net notation of places.

ProNet has a combination of features which make it rare among modelling techniques. It was designed explicitly for software process modelling, and its entity classes are tailored to this goal. In addition, it was developed so that there could be a direct and unique correspondence between graphical process modelling in ProNet and an equivalent symbolic enactment model. Finally, ProNet provides for version management within a process definition / enactment context and for persistency of entities generated during enactment.

ProNet diagrams (see Figure 2.4) are based on a modified entity-relation model, in which the entities fall into one of eight classes;

1. Activities. The existence of entrance conditions / products allows activity initiation, which is then responsible for generating exit conditions / products.
2. Products (e.g. a source code file) may be the result of some activity internal to the model.
3. Conditions can either be required to initiate an activity or result from an activity.
4. Composites are boolean combinations of conditions, products or agents.
5. Agents are specific entities which perform activities.
6. Roles are abstractions (i.e. super-classes such as reviewer or developer) of the agents concept.
7. Stores allow for persistence of instantiated entities.
8. Constraints are policy restrictions imposed on the performance of an activity.

A ProNet model consists of a network of entities linked by a standard set of relationships. ProNet allows for a hierarchical decomposition of the process model, i.e. a process diagram at one level may contain entities which are expanded to show lower level of detail. From the graphical process model, a set of production rules can be

derived through which the process can be executed. Translating the ProNet model to an enactable form is the subject of current work.

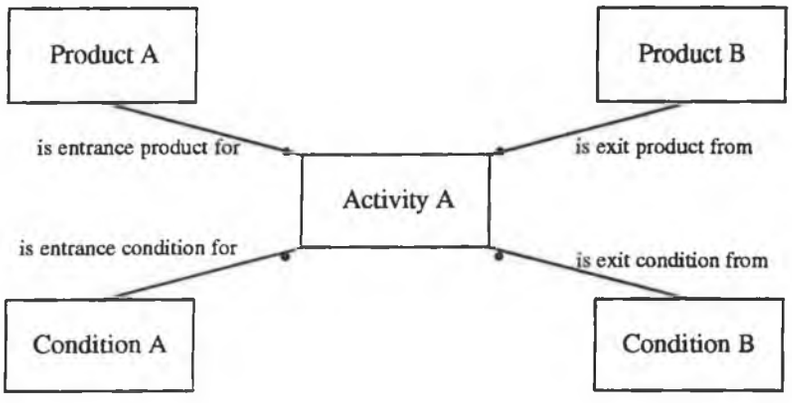

Figure 2.4 Basic representation of a process element

[Chr93] findings on ProNet state that;

> *"...its graphical notation can be understood quickly by individuals not familiar with it or other modelling notations, and it is an excellent communications tool for extracting knowledge from the expert on the project's process."*

2.1.6 Statecharts

[KH87] details the use of the STATEMATE system to develop software process descriptions and to analyse and simulate their behaviour. This approach has also been put forward in [Kel91] [KM93]. STATEMATE offers process model builders a set of three distinct but interrelated viewpoints with which to model a real-world system. Taken as a whole these three perspectives cover the traditional "Who, What, When and How" of a process. These viewpoints are;

- Functional - What is to be done; represented by Activity Charts.

- Behavioural - When and How it is done; represented by Statecharts.
- Structural - Who and Where it is done; represented by Module Charts.

With STATEMATE, a process is described through graphics and textual form. The graphics are used to describe the three viewpoints as activity charts, statecharts, and module charts (as above). The textual forms are used to describe connections between the views, formal definitions, and informal descriptions. In addition to providing these mechanisms for representing the software process model, STATEMATE provides a number of powerful capabilities in the area of analysis and simulation.

The graphical languages utilised for the three viewpoints are quite similar and are based on higraphs. They utilise two major components: named boxes and directed lines. The named boxes represent activities, states, and modules, respectively, in three types of charts. Hierarchical decomposition is represented by nesting within the same diagram. This is in contrast to most other toolkits, which use separate diagrams to depict different levels of detail. Directed lines represent information flow in activity charts and module charts and represent state transitions in statecharts.

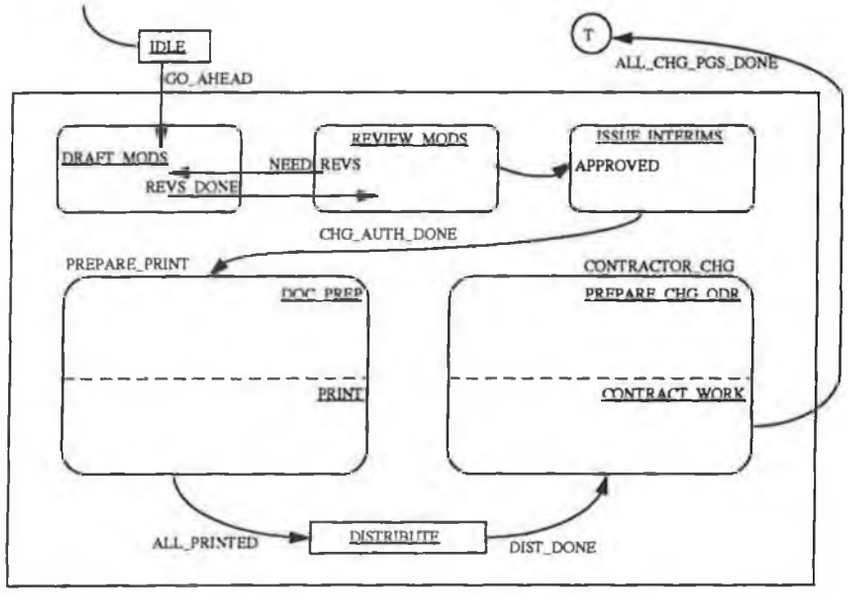

Figure 2.5 Statechart - Top Level Process

Figure 2.5 presents a top level view of a statechart representing the behaviour of a modification process. The line starting in the upper left-hand corner leading to IDLE is called the default transaction and indicates where the system starts off. Thus we have defined the process to begin in an IDLE state. When the GO_AHEAD event occurs, the process moves from the IDLE state into the DRAFT_MODS state, meaning draft modifications. The process continues in this manner, with states changing as events trigger until the ALL_CHG_PGS_DONE event triggers and the process moves to a state containing a T, which indicates the process has terminated.

2.1.7 Visual Process Language

Visual Process Language (VPL) is a formal process modelling and programming system designed to provide both visual representation and detailed descriptions for enaction of software development processes [Sib91]. VPL models processes as objects flowing through a graph. Associated with each VPL graph are two tables, which contain objects and roles. Together these items constitute a VPL program. An object in a VPL program represents all the artifacts associated with the currently active work assignment. A role is a label that is attached to every system user to indicate the functions that user will perform in a VPL program.

A VPL model of a software process is a directed graph of nodes and edges. There are nine kinds of nodes: Start, Finish, Procedure, Task, Decompose, Recompose, Split, Merge and Branch (see Figure 2.6). Start and Finish nodes represent the points at which objects enter and exit a program. A Procedure node is a convenient way of abbreviating a sub-graph of nodes. A Task node represents an action performed on the object either by an automated tool or by a tool and a user. An object entering a Decompose node becomes a family of sub-objects, each of which represents a portion of the original object. This family may be changed back into a single object in a Recompose node. An object exits a Branch node along only one of its output edges. Finally, an identical copy of an object entering a Split node emerges along each of the

output edges. Each of these copies is typically modified in some unique way prior to accumulating at a Merge node. Eventually all but one of the modified copies are deactivated, and the remaining copy moves on to the next node.

VPL programs are enacted using four primitive mechanisms: Creation, Advancement. Deactivation and Reactivation. An object is created at a start node by system users to initiate a project. Under most circumstances, when the action associated with a node is completed, an object will advance to the next node in the graph. An object may be deactivated by authorised users when it is no longer needed, at which point its state and files are archived by the system configuration manager. If the deactivated object is required at a later date, it may be extracted from archives, and it will continue progression through the process.

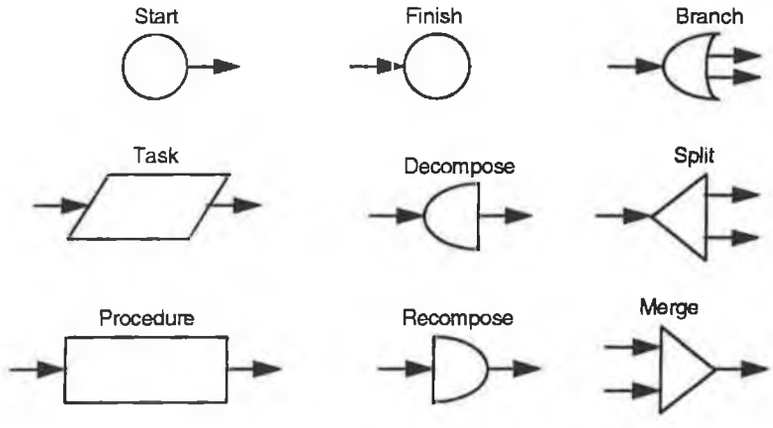

Figure 2.6 VPL Symbols

Many complex control patterns can be modelled using the four mechanisms and nine node types included in VPL. Associated with each node in a VPL program is a set of conditions which determine when an object residing at that node is ready to advance. These conditions might be determined by the results of tools, by off-line human decisions or a combination of several factors of both categories. When conditions are met, the objects state is copied to archives, and it advances to the next node. An object may be backtracked by deactivating it in its current form and reactivating a former version. As objects progress through a VPL program, they are archived at each node, for later reference if necessary.

While VPL models are easily created and edited visually, it is more efficient for storage and enaction, to represent them internally as text. The transformation between the two representations is performed by the environment. This environment has been prototyped in SunView on a Sun SPARCstation [SSW92b].

VPL is a suitable language for process modelling as it places few constraints on the tools or the processes itself and allows users the flexibility to develop creatively while controlling process shortcuts and hence minimising the possibility of later problems. It combines a conceptually intuitive visual modelling language with a powerful and flexible process programming language [SSW92a].

VPL is examined in further in Chapter 3 as a basis for building E/VPL.

2.2 Example of Software Process Environments

Attempts to automate the software process are motivated by our needs to improve the quality and productivity of our work. When we can reduce a task to a routine procedure and then mechanise it, we not only save labour but also eliminate a source of human error - which is the most effective way to improve productivity [Hum89a]. It is this need which has lead to the development of process support environments (cf. section 1.1), which is the focus of this section.

In this section I am concerned with the use of process modelling techniques within process support environments. There are many names given to such styles of environments, such as Integrated Process Support Environment (IPSE), Integrated Coding Support Environment (ICSE), Process Guidance Systems, Computer Assisted Process Enactment (CAPE) and Process Centred Environment (PCE), but we will settle on the general definition of a Process Centred Environment as;

"...a SEE with a process engine that manages process knowledge and guides users." [AO93]

Before we can progress to examine some SEEs, we must first examine their general architecture and characteristics.

[Hum89a] lists the basic characteristics of a SEE as;

- The environment must be easy to use.
- We must be able to customise the environment.
- The architecture must be open and capable of evolving with the needs of the project.
- It must provide for strict enforcement of a liberal process.
- It must have a comprehensive conceptual schema that encompasses a database, process data, tool interfacing, and environment evolution.

Penedo and Riddle [PR88] identified four layers of support in a SEE and use the expression "virtual machine" to describe the aggregation. Moving from lowest to the highest, these are the hardware and operating system layer, the environment support layer, the tool / capability layer, and the project user support layer. Few tools, with the exception of Arcadia [TBCO88], structure themselves in such layers. Most focus on the tool and user-support layers.

A fundamental difference among process modelling approaches and guidance systems is whether they support modelling in-the-small or modelling in-the-large [Tul86]. Tully describes this as a distinction in perspective rather than a distinction among project size. The in-the-small approach focuses on fine-grained, easily automatable activities of a short duration (e.g. compiling code), whereas in-the-large targets the problems involved in guiding teams of people who work together to achieve a common goal (e.g. resource allocation). Fernström [Fer93] categorises the coupling between a process guidance system and the corresponding real-world process according to the

following four levels, the influence of human creativity is progressively restricted between levels;

1. Loosely coupled: No connection exists between the system and the real-world. The system relies on the user to report any changes in process state.
2. Active support: Access to tools and data is partially automated. This allows guidance systems to perform simple tasks.
3. Process enforcement: Access to tools and data is totally controlled. Users are able to access only those resources that are directly necessary for the current task.
4. Process automation: No human creativity is required to perform the process and the process guidance system can accomplish the work without human intervention.

A number of SEEs are discussed in the following sections. They are meant to give an appreciation for the work being done in this area. It is by no means a comprehensive list of all SEEs.

2.2.1 ALF

In ALF [OZG91] a software process model is described as a hierarchy of MASPs (Model for Assisted Software Processes). Each MASP describes a part of the software process model, which in turn, can be detailed by other MASPs and so on. This permits a description at different levels of abstraction.

A MASP is a generic description that must be instantiated before being executed. The instantiation need not be completed before enaction begins. Instead, instantiation and enaction may interleave so that the part of the development that has already been executed may be taken into account to instantiate a further part.

Each MASP is described as a 6-tuple (Om, Op, Ex, Or, Ru, Ch). Om, the Object model, provides a conceptual data model based on ERA (Entities, Relationships and Attributes). Ex is a set expression, specified in a first order based logic language, that is used to describe operator types (Op), rules (Ru) and characteristics (Ch). Operator types describe the semantics of software process activities in terms of pre and post conditions. Rules define the possible automatic reactions to specific situations arising during the software process. Characteristics specify constraints on the process states; if they are not satisfied, they raise an exception condition. Finally, Or is a set of orderings, specified using path expressions in parallel, alternatively, or sequentially.

One of ALF's main goals is to provide assistance during software development. The user receives guidelines about what to do next, how a certain process works, or how to perform a certain action. In addition, explanations are given when the invocation of an operator is rejected, when the system takes an initiative on its own and when an operation invocation would violate the characteristic.

2.2.2 Articulator

Articulator [MS90] uses knowledge engineering techniques to understand the process and is focused on managing, or articulating, conflicts as they arise during process enaction. It uses an object-based language to specify process models, a knowledge-based mechanism to answer user queries about models and behaviours and a simulator to test process model behaviour. Processes are simulated using a state machine wherein each state is defined as a snapshot in time (i.e. multiple identical states in a process history are possible).

[MS93] details the construction of a prototype graphical user interface (GUI) for Articulator, which can be used to visualise and animate an instance of a software process model (SPM). Their Process-Based user Interface (PBI) is coupled to another

computational facility in which SPM instances developed with articulator can be automatically transformed into process programs.

2.2.3 EPOS

EPOS [ABGM92] [JC93] is a process support environment that offers a process manipulation language (PML) call SPELL (Software Process Evolutionary Language), an initial process schema, and a set of process tools. In EPOS, the internal process model is a network of activity descriptions (tasks), being linked to descriptions of other tasks, products, tools, and roles. The activities interact with each other and with tools and humans.

The process model schema is represented as a set of entity and relation classes that constitute template fragments. The meta-model part of the process schema is represented as a set of meta-classes. The instantiated and enacting process models consist of instances representing external process elements. The EPOS model fragments are meta-classes, classes, and instances; both entity and relation. The main process tools operating on the above models are; a process model manager, an execution manager (process engine), an artificial intelligence planner, and EPOSDB, a versioned object-oriented database.

The EPOS process environment possesses a very rich variety of features, borrowed from nearly all approaches that are present in literature, and seems to allow a very dynamic and flexible management of processes. However, it is quite difficult to get an overall picture of the system because it combines numerous heterogeneous constructs and facilities, without providing detailed formal semantics.

2.2.4 Marvel

Marvel [KFP88] was the major focus of the Programming Systems Laboratory at Columbia University from 1987 until 1993. Its goal was to develop a kernel for a process-centred environment that would guide and assist a team of users working on a medium-scale software development effort.

It is a knowledge-based programming environment that assists its users during the implementation, testing and maintenance of software projects. Users work on their code artifacts through a Marvel interface, which allows the system to be aware of the user's activities. Processes are modelled using the Marvel Strategy Language (MSL) as strategies (rules). Opportunistic processing is used, meaning that work is performed when the system detects that it can be done based on rule preconditions. Using these rules to match user activities, Marvel can automate small, formal activities such as compiling. Marvel is limited to controlling processes involving tool invocations.

A project administrator specifies a software process model in terms of rules. The rules are classified in three different sets: the project rule set that describes process specific issues, the project type set which is used to specify the data with object-oriented classes and the project tool set which describes the interface with external tools. The project rule set is composed of two kinds of rules; activation rules and inference rules. Activation rules control the initiation of development activities, and typically involve the invocation of a tool. Inference rules only define relations among attributes of objects. Rules can be executed by forward chaining or backward chaining. If the user wants to execute a rule whose condition is not satisfied, backward chaining is applied to fire other rules whose effect might satisfy the condition. The result of backward chaining is either the satisfaction of the action or a notification to the user that the command cannot be executed. If the condition is satisfied, the activity is initiated and, after it terminates, one of the rules is asserted. This may enable the execution of other rules causing forward chaining.

Marvel version 3.1 was released in March 1993, and consists of about 154,000 lines of C, yacc and lex code. It supports a choice of three user interfaces - XView, Xlib and tty - and runs on Sun SPARCstations with SunOS 4.1.3, DecStations with Ultrix 4.3, and IBM RS6000s with AIX 3.2. The Marvel 3.x series has about 40 licences at time of writing this thesis.

2.2.5 MELMAC

MELMAC is described in [DG90] [EG91] [Gru93]. Processes are modelled, analysed, and enacted using a comprehensive internal representation named FUNSOFT nets (cf. section 2.1.2). All necessary information about the processes is stored in the internal representation.

Views are used for user-level representation, specifically; object types, activities, data flow, staff responsibilities, and feedback are all defined using special views onto the internal representation. Work done both on and off the computer can be described and supported. Pre and postconditions on activities describe constraints on the work. The network allows the environment to analyse as well as execute a process model. Users are guided in their work through a common user interface.

MELMAC provides a graphical editor, to create and modify FUNSOFT net descriptions, as well as simulation of the net. Within the MELMAC environment all the information is stored in the object management system GRAS. The GRAS data model is one of attributed, directed, acyclic graphs. Accordingly, software process models are stored as graphs. All software process relevant data is stored in one common project database. The MELMAC environment has been prototyped in C on SUN workstations under SunOS 4; and currently consists of about 40,000 lines of code.

2.2.6 MERLIN

MERLIN [EJS91] is a software process programming language, where a rule based technique is used to build a knowledge base describing the software process. Rules and

facts in the knowledge base may be interpreted in two forms; backward chaining and forward chaining.

Backwards rules and facts are given Prolog-like notation and are interpreted in a Prolog-like manner. The backward mechanism is exploited to select the roles and the activities a user may perform. It is also used to collect information in order to answer queries on the process state. Forward chaining instead is applied when explicit guidance is provided by the system. The rules being interpreted by forward chaining consist of a precondition, a list of activities and a postcondition. Since rules and facts may be dynamically inserted and deleted from the knowledge base, the model exhibits great flexibility in representing changes that occur during the process execution.

MERLIN supports the basic abstractions of any software process, such as activities. software objects, roles, and resources. However, at present time, it lacks structuring mechanisms for large rule sets. Parallelism is recognised as a central research topic, but currently is not supported.

Users interact with the system via a hypertext interface that shows the user's working context, including all objects for which the user is responsible and defined actions for those objects. The rule-base is thereby hidden from environment users. Persistent storage of objects in the MERLIN system is accomplished through use of a database scheme that is specialised for storing rules and which facilitates rapid access to and selection of rules.

Currently work on MERLIN focuses on transaction management, which is based on the combination of optimistic and pessimistic protocols. GRAS, as the currently used database, does not offer very sophisticated facilities for storing and retrieving fine-grained information items, thus the MERLIN system is currently being ported to a fully object-oriented system. Further effort is being directed towards the creation of a process engineer's support environment [PS92].

2.2.7 Process WEAVER

Process WEAVER [Bou93] is a modelling and enactment system (or process support system), which has been designed in order to be integrated with existing development environments, to transform them into process centred environments.

Process structure and product flow is represented using graphical editors and process behaviour is represented using a Petri net derivative. The internal modelling language is completely hidden from users by views; these are tools that display and manipulate the internal representation. At enaction time, the "agenda" tool within the system presents tasks to process participants and mediates interactions with tools and information. This tool allows its users to work both with and without process guidance and to create, modify, and delegate tasks to other users. The system has extensive support for interfacing to CASE and other software tools within the supporting computer system.

Process WEAVER has been designed to ease the construction of integrated process-support environment. It is a means to introduce process support within existing development environments. Its architecture thus conforms to the necessary requirements it supposes; openness of the system, integration mechanisms, etc.. It is decomposed into a set of tools which are organised around an integration and communications system, called the software bus. All its components have an application program interface (API) which allows them to export the required services. The software bus is based on a broadcast message server which allows tools to send and receive requests to and notifications about the exported services.

2.2.8 The √iewer

The √iewer [NF93] [NFK93a] is a VOSE (Viewpoint-Oriented Systems Engineering) support environment implemented in Objectworks / Smalltalk release 4. The notion of ViewPoints is central to this environment. [NFK93b] defines ViewPoints as;

"...loosely-coupled, locally managed, distributable objects, encapsulating representation knowledge, development process knowledge and (domain-specific) specification knowledge."

This framework uses ViewPoints to describe system development participants, their roles in the development process and their views of the problem domain. ViewPoints encapsulate systems development knowledge in five separate slots;

1. Style - the notation or representation style used.
2. Work plan - the development actions and strategies that use the notation.
3. Domain - the problem domain.
4. Specification - the actual specification.
5. Work record - the development history.

Each ViewPoint is associated with a particular development participant (role) called the ViewPoint owner. The ViewPoint owner is responsible for enacting the ViewPoint work plan to produce a ViewPoint. Clearly, a number of ViewPoints may employ the same style and the same work plan, to produce different specifications for different domains. It is therefore possible to define reusable ViewPoint templates in which only the style and work plans are elaborated.

The √iewer uses a very basic approach to tool implementation and integration. Currently all the ViewPoint template support tools are implemented in Smalltalk - with the architecture allowing for their easy addition, modification and extension. Standard abstract classes are available to get skeleton tools operational quickly.

The √iewer is a large prototype Smalltalk application implementing over 45 classes. Work is currently underway to improve consistency checking and method guidance, and to provide hooks to ViewPoint support tools written in languages other than Smalltalk [NF93].

2.3 Summary

Having considered the variety of representations and SEEs that have been implemented in order to automate the software development process, one could be forgiven for thinking that the area has been saturated, but not so, there are still many problem areas which need to be addressed.

I consider that most of the SEEs currently available do not go far enough to provide what is really needed, and in some respects, go too far to provide what is not needed. Automation is not a solution, it is an aid to a solution. We cannot expect to replace creative programmers and managers by automated tools, but we can certainly develop far more capable and flexible support tools. After all, an environment can only be expected to support an effective process, not create it.

One of the most important issues is the notion of control versus support. So, in implementing any environment we must recognise that no one person is intelligent enough to define precisely what should be done in a typical medium to large scale software project. We must be careful that the imposition of constraints and controls always leaves room for exceptions and escapes. Many of the environments considered in the previous section provide an integrated framework for the development of the software and not just the software development process or model. They control the users every action with an "Intelligent Assistant", first checking to see if the user has authority to perform the task and then presenting the only possible way forward and then, if that is not enough, they whisk away the code just assembled and have it compiled and checked for errors, all of which is recorded in a project history. These Orwellian style "Big Brother" environments are in my opinion not a practical way forward.

There are human factors which must be taken into account [Sta93];

- Strictness: How strictly should a model be enforced. Some people might feel oppressed by strictly enforced models, whereas other feel lost without strong guidance.
- Granularity: If a model is too fine-grained, it becomes unmanageable due to its size. A too coarse model on the other hand might be too abstract and will not provide sufficient guidance.
- Creativity: Every development process contains numerous creative subtasks. Even today it remains unknown, how creative processes can (or should) be

modelled. Granularity and strictness will definitely have strong influence on creativity.

- Communication between humans is often informal. A lot of process and project knowledge is transmitted by inter-personal and informal communications, which cannot be adequately represented by modelling formalisms.
- Argumentation and decisions are important sources of knowledge within the ongoing development process.
- Productivity: By what means can the productivity of the development process be enhanced?

Many of the problems with the environments outlined above are human-orientated. People do not like environments that take control of their work and guide them to the extent that they loose all control and decision making. Also, most of the environments provide a formalism which is difficult to understand and requires a large amount of special training to program. In the light of these core problems, I propose the creation of a Process Centred Software Development Support Environment, which would be capable of guiding software developers in their work, answering questions like: "Where am I?", and more importantly, "Where should I go next?". This environment should be easy to use and more "people friendly", allowing users to be "gently" guided in their work and providing a simple, yet powerful, formalism to describe processes.

This leads to the question: On what style of formalism should such a support tool be based? As mentioned in section 2.2, textual process modelling languages are error prone and create process models which are difficult to grasp conceptually. Therefore I would propose the use of a graphical formalism to represent the software process model. In order to make an effective and intuitive process modelling tool, I suggest using a formalism which will combine flexibility and robustness with ease of use and speed of learning. It is for these reasons that I would propose such a support tool be based around the notion of a graphical formalism such as VPL, which combines expressive power with ease of use and understanding.

In Chapter 3 we will examine VPL in more detail, in order to assess its suitability as a software process modelling language and its application to a process centred software development support environment.

CHAPTER 3 - VPL - A CLOSER LOOK

3. Introduction

Visual Process Language (VPL) [Sib91] was developed at Royal Military College, Ontario, Canada. VPL is a second generation Integrated Project Support Environment designed to permit the description and enaction of software process models.

In this Chapter we will examine in detail the notation employed by VPL, the architecture of the VPL environment and further investigate the VPL enaction paradigm, using an example taken from a VPL case study. Having further considered VPL, we will outline some of its weaknesses and limitations and propose possible alternatives and extensions.

3.1 VPL Notation

VPL models processes as objects flowing through a graph. Associated with each VPL graph are two tables, which contain objects and roles. Together these items constitute a VPL program. An object in a VPL program represents all the artifacts associated with the currently active work assignment. A role is a label that is attached to every system user to indicate the functions that user will perform in a VPL program.

A VPL model of a software process is a directed graph of nodes and edges. There are nine kinds of nodes; Start, Finish, Procedure, Task, Decompose, Recompose, Branch, Split, and Merge (see Figure 2.6);

1. The Start node represents the point at which objects enter a program.
2. The Finish node represents the point at which objects exit a program.

3. A Procedure node is a convenient way of abbreviating a sub-graph of nodes.
4. A Task node represents an action performed on an object either by an automated tool or by a tool and a user.
5. An object entering a Decompose node becomes a family of sub-objects, each of which represents a portion of the original object.
6. This family created by a Decompose node may be changed back into a single object in a Recompose node.
7. An object exits a Branch node along only one of its output edges.
8. Identical copies of an object entering a Split node emerges along each of the output edges.
9. Each of these copies of an object created by a Split node is typically modified in some unique way prior to accumulating at a Merge node. Eventually all but one of the modified copies is deactivated and the remaining copy moves on to the next node.

3.2 VPL Architecture

The architecture of VPL is composed of five main components (see Figure 3.1), as follows;

1. User Model: Two of the purposes of software process enaction are to delegate tasks and to track responsibilities. VPL accomplishes these goals by assigning to each person involved in a project one of four roles [Tre90]; User, Supervisor, Manager, Process Programmer.
2. Tool Model: The tool interface in the VPL environment consists of a command line string sent to a teletype window.
3. Data Model: This manages multiple versions of user data files belonging to multiple versions of active and inactive objects.
4. User Interface: This environment has been prototyped in SunView on a Sun SPARCstation.

5. Process Model: This brings together the process program files and object files within the enaction mechanism.

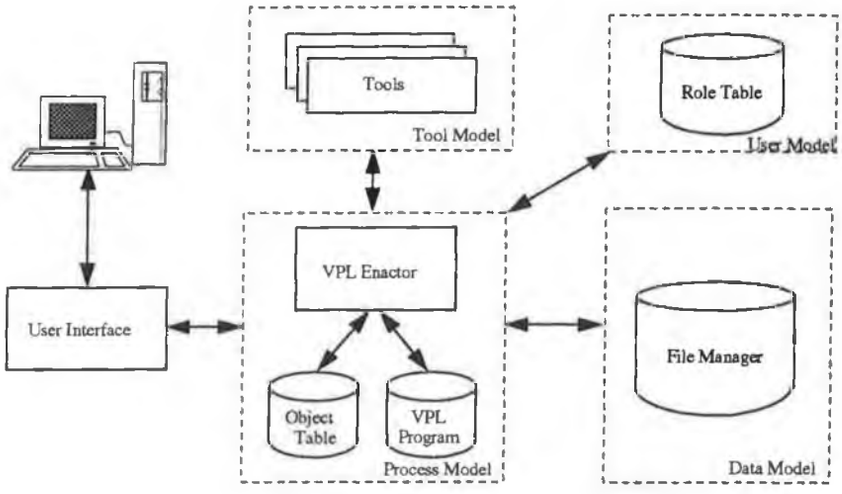

Figure 3.1 VPL Environment Architecture

3.3 The VPL Paradigm

The VPL enaction paradigm combines some of the features of the object-oriented paradigm, Petri nets and logic flowcharts. Enaction is accomplished through four primitive mechanisms; Creation, Advancement, Deactivation and Reactivation. An object is created at a start node by system users to initiate a project. Under most circumstances, when the action associated with a node is completed, an object will advance to the next node in the graph. An object may be deactivated by authorised users when it is no longer needed, at which point its state and files are archived by the system configuration manager. If the deactivated object is required at a later date, it may be extracted from archives and will continue its progression through the process.

Associated with each node in a VPL program is a set of conditions which determine when an object residing at that node is ready to advance. These conditions might be determined by the results of tools, off-line human decisions or a combination of several factors of each category. When certain conditions are met, the object's state is copied

to archives and it advances to the next node. An object may be backtracked by deactivating it in its current form and reactivating a former version. As objects progress through a VPL program, they are archived at each node for later reference if necessary.

In software development, the results of one work assignment are normally used as the starting point for the next work assignment. This normal flow, called linear control. breaks down into the following steps; when an object enters a node, the action associated with that node is performed on it and it exits the node; the nodes action may change the objects state or even the number of objects; if the objects state has changed. the entire object or set of objects is recorded and appropriate messages are automatically sent to all personnel concerned.

Non-linear control sequences are performed using object deactivation and reactivation. Deactivation is the process of removing an object from the active object table, thus suspending work in that area. During deactivation, a copy of the object is retained in the inactive object table. Reactivation is the process of extracting an object from the inactive object table and placing it in the active object table. Using these mechanisms. the following non-linear sequences can be followed;

- Backtracking is accomplished in the VPL paradigm by deactivating an object and reactivating a former version of the same object. Since the past history of every object is stored in the inactive object table, an object may regress to a former state at any time. After backtracking, an object may also return to a more advanced state.
- Races are a type of concurrency where the first stream to finish (or the best result) is kept and all the others are discarded. They are facilitated in the VPL paradigm by duplicating an object in the object table. The duplicate objects may be assigned to different people or teams. All but one of these parallel strains may subsequently be deactivated, or possibly multiple versions of the final product may be produced.

- Concurrent activities on separate streams of a project are permitted through the use of decompose and recompose nodes.
- Cancellation of a project is performed by deactivating all objects associated with it. Since the objects are kept in the inactive object table, they may subsequently be reactivated, which "un-cancels" the project.

Since the timing of these non-linear sequences is not generally known a priori, it would be futile to try to design them into each particular process program. For this reason, deactivation and reactivation are globally available to users (within permission constraints) through the enaction paradigm, rather than being explicitly represented in the VPL program.

An object in a VPL program represents all of the artifacts associated with an individual work assignment. These artifacts include files, participant names and state variables. While the files are actually managed by the data model (cf. section 3.2), they are identified by keys stored in each object. Participant names are the names of the people fulfilling the various roles. State variables are fields used as flags to store information about the object, such as the current program and position of the object (i.e. which program, procedure and node it is in), the current actor (user role currently operating on the object), decomposition history and so on.

VPL implements objects as follows;

- Active objects are stored in the object table: They are represented as a set of values corresponding to the object name, file identifiers, participant names and state variables; The VPL object table is analogous to the process table in a multi-tasking operating system.
- When an object is deactivated, a copy is placed in the inactive object table: Inactive objects may be retrieved and inserted into the active object table. This might be done for a number of reasons; E.g. for an audit, to backtrack an activity, or to initiate a race.

3.4 A VPL Example

A prototype of VPL and an environment with the architecture described in section 3.2 was implemented in Royal Military College, Canada, under SunView on a Sun SPARCstation [Sib91]. This system was evaluated at the Aurora Software Development Unit (ASDU) in Greenwood, Nova Scotia, Canada [SSW92a]. ASDU is responsible for the maintenance of over five MLOC's used to support the CP-140 Aurora patrol aircraft and its support system.

Figure 3.2 shows the root procedure of the software change process used at ASDU, represented in VPL. A project manager wishing to initiate a software change creates a new object in the start node of this program using a specialised tool which is part of the environment. The new object consists of a rough statement of work for the software change. When it has been created, it advances automatically to the first node where human intervention is required. In the root procedure shown, the object advances to the *generate software change request* (SCR) procedure.

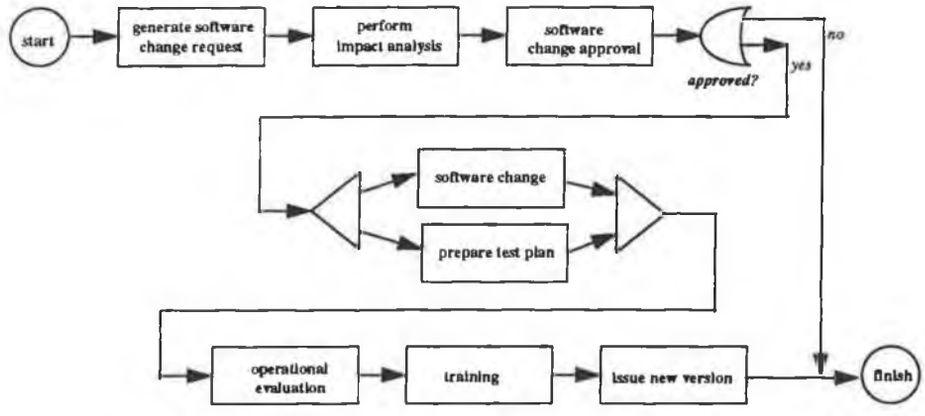

Figure 3.2 ASDU software change process: root procedure

The nodes compromising the *generate* SCR procedure are shown in Figure 3.3. The object passes through the initial *start* node of the *generate* SCR procedure and stops at the *enter and categorise* SCR task node. The *enter and categorise* SCR node in Figure 3.3 might encapsulate a text editor, or it might provide users with a more specialised tool. In any case, the object residing in the *enter and categorise* SCR task node

appears as a job available to a person who is required to perform work. When the person selects the job, the appropriate tools are automatically invoked with the necessary data files. When the job is complete, the object advances to the next node.

The next node in Figure 3.3 is a branch node (category) which in this case is fully automated. The category assigned to the SCR in the *enter and categorise* SCR node determines which path the object follows through the remainder of the procedure.

The SCR continues through the first three procedures of the root procedure shown in Figure 3.2 and if approved, encounters a split node. At this point, the object "owns" many files which form a record of all the work performed to this point. The split node produces two copies of the entire object and sends one along each of the output edges to the *software change* and *prepare test plan* procedures. Actually the data files are not duplicated, but pointers are retained by each child object until such time as the files are changed. The two child objects are likely to be jobs for two different users. Thus the software test plan and the development itself proceed concurrently and independently, based on the change request and impact analysis. After the software has been changed and the test plan prepared, both child objects encounter the merge node. The single object constructed in the merge node is an aggregate of the data files of the two input child objects, and contains both the changed software and the test plan.

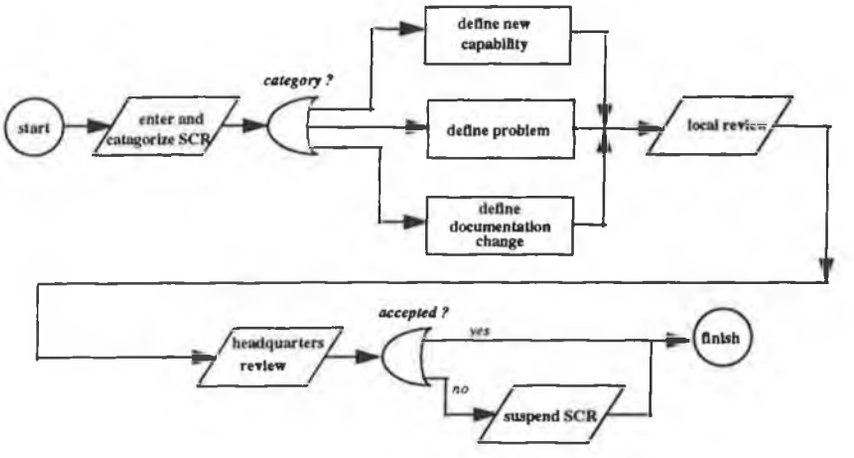

Figure 3.3 ASDU generate SCR procedure

In Figure 3.2, if the software change is not accepted during the operational evaluation, the SCR will be either cancelled or reworked. If it is cancelled, all objects associated with the SCR are deactivated. If it is reworked, each object will be moved back sufficiently far to allow the perceived problem to be corrected. The latter action is facilitated because a copy of each object was archived every time it emerged from a node.

The *software change* procedure of Figure 3.2 contains a subprocedure called *implement*, which is shown in Figure 3.4. An SCR object which enters at the start node passes through the *detailed design* procedure and enters the *break into modules* (decompose) node. In this node, the detailed design is broken into its component modules - that is each module that is affected in the detailed design appears as a separate object. Each of these objects passes through the *code and debug* procedure where the modules are edited and compile errors are removed. Finally, the objects collect at the *collect modules* (recompose) node and, when a required set is coded and debugged, the SCR is ready for linking and integration testing (in some other procedure). Any adjustment in the module decomposition or in the specifications of modules that have already reached the recompose node requires backtracking to the decompose node. When the process moves forward again, objects that are not affected by the adjustment will be retrieved from the archive.

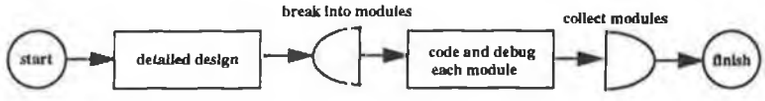

Figure 3.4 ASDU implement procedure

After the SCR object has passed through the entire program shown in Figure 3.2, it will stop at the *finish* node where it is deactivated and retained only in archives. This is the normal default end point for all objects entered into the system.

This example was used in [SSW92a] to find weaknesses and flaws in the existing informally described process model and in clarifying the model. As a result of this

example, participants considered VPL to be a potentially useful process modelling and process programming language.

3.5 Constraints and weaknesses of VPL

In this section I will explore some of the limitations and constraints of VPL, thus leading to the motivation for making changes to VPL. I have broadly classified the shortcomings of VPL under the following headings;

- **Terminology**

  The impression given by some of the VPL terminology may not convey the correct meaning: The difference between a task node and a procedure node is not immediately obvious and a VPL object might be misconstrued as having some association with the object-oriented notion of objects, whereas in fact it does not. These may be regarded as superficial points, but the subtlety in terminology becomes apparent when a new user is introduced to VPL.

- **Node redundancy**

  Split and Merge nodes were not part of the original work carried out on VPL, but were added as an afterthought. There appears to be a large element of redundancy between the ideas of Split and Merge nodes and those of Decompose and Recompose nodes and I do not consider it necessary to have both sets of notions, but instead to redefine broadly the notion of split and merge. Also, the VPL environment does not enact decompose and recompose nodes, object duplication and merging must be done manually by the process programmer through multiple reactivations.

- **Object implementation**

  VPL regards objects as dealing with "an individual work assignment"; This is in my opinion an incorrect interpretation of the artifacts associated with a software

project. Objects are in reality large complex items, which are usually shared among project participants, so to treat an object as associated with an individual project participant is too simplistic. Also, when an object enters the ultimate Finish node in a process program, all objects associated with that program are deactivated and archived. This would also appear to be a simplistic approach to object enaction, objects may well persist longer than the project process life time or further information may be required about objects at a later stage. For example, a project budget (object) may be required for review long after the project has been completed.

- **Enaction Paradigm**

  An object automatically progresses forward from a node to its successor when the objects state variables meet all of the post-conditions (which are stored as a boolean expressions in the node). This automatic advancement is, in my opinion. a crude form of object enaction, as is does not allow for exceptional circumstances or other possibilities. One characteristic which may become a limiting factor is that the branching and exit condition strings, and state variable sets are small, which means that the environment can only track a small number of object attributes which will in turn affect control flow. Also, the recording of a process history is achieved by archiving a copy of an object after its emergence from each node. This is in my opinion an elementary method of recording a process history.

- **Architecture**

  The main problem that I see with the VPL architecture is that the many of the main functions of the environment are shared across all of the components. For example, the object information, role information and all sundry data involve the interfacing of three separate system components. This division leads to a system with no central storage of a process model, rather a scattered representation of the various parts of the model across the architecture. There is also, no clear division between the two major functions of the environment; creating (or defining) the process program and enacting that program.

- **Environment**

  The style and quality of the user interface is poor (although this can be substantially accounted for by the fact the system is an academic prototype and also the availability of programming tools has increased dramatically over the last five years). The tool interface in the VPL environment consists merely of a single command line string sent to a teletype window, so where more than one line is required to invoke a tool, a batch file has to be used. This is not an acceptable form of tool integration. Also, the file management system (implemented as flat files) employed by the system, is inadequate for the effective management of large volumes of data and it leads to a massive number of files and complicated directory structures.

3.6 Summary

In this Chapter we have examined in detail the notions behind VPL, its architecture, notation and enaction paradigm. Finally, we have looked at some problem areas with VPL in its current state.

Following from this, in Chapter 4, I will attempt to overcome at least some of these problems by making extensions and enhancements to VPL. This will be done in the light of currently perceived requirements of process modelling systems and of the problems mentioned above.

CHAPTER 4 - DESIGNING E/VPL

4. Introduction

In this section I will describe E/VPL. I will start by describing the formalism, based on VPL with modifications to overcome the problems outlined in the previous chapter. Then I will detail an architecture for a computer environment to support the design, maintenance and enaction of E/VPL process models.

4.1 Extending VPL

E/VPL is a formal programming language designed to visually represent and permit enaction of the software development process. It is based on the concepts of VPL, with enhancements aimed at overcoming the problems outlined in section 3.5. It is composed of a graphical formalism to represent software processes and a software environment to support enaction of the process model.

An E/VPL model of the software process conforms to standard VPL rules. It is a directed graph of nodes and edges which satisfies the following constraints;

- The graph must be fully connected.
- Each node must be one of the valid set of seven node types.
- The graph must contain exactly one start node.
- The graph must contain exactly one finish node.

A process model described using E/VPL consists of a graph and data about the process, including process descriptions, user roles, resource information and information collected during process enaction (e.g. metrics data).

4.1.1 E/VPL notation

The notation employed by VPL (cf. section 3.1) has been broadly maintained with some minor alterations to satisfy the arguments of section 3.5.1. The E/VPL notation contains seven node types (see Figure 4.1);

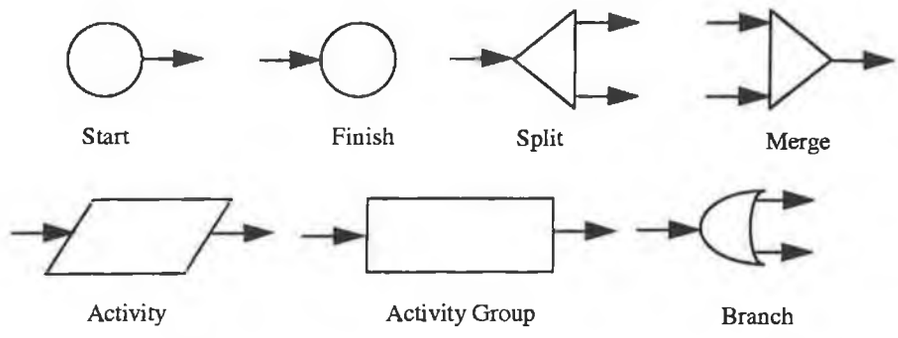

Figure 4.1 E/VPL symbols

1. The Start node represents the point at which artifacts enter a process program.
2. The Finish node represents the point at which artifacts exit a process program.
3. An Activity node represents an atomic action performed on an artifact (by either an automated tool or a user), or an action performed off-line, without affecting an artifact.
4. An Activity Group node is a convenient way of abbreviating a sub-graph of nodes. It is in itself a process program conforming to standard E/VPL rules.
5. A Split node represents the splitting of a work stream into two or more parallel streams of work, with the artifacts of the split node being available to all output streams.
6. A Merge node is the point at which two or more streams of work may be synthesised into a single work stream. The artifacts available at the merge node are a synthesis of the artifacts of the input streams.
7. A Branch node represents a decision point in a graph. Artifacts may exit along one of the output streams only.

The following modifications have been made to the original VPL nodes;

- Task node;
  - The Task node has been renamed an Activity node, to avoid conflict of meaning between it and a Procedure node.
  - The scope of an Activity has been broadly redefined to include non-automated or off-line activities performed by process participants, which may or may not affect process artifacts.
  - An Artifact node does not have to represent the invocation of a tool, but rather work done by a human(s) or a tool(s).
- Procedure node;
  - The Procedure node has been renamed an Activity Group node, to avoid conflict of meaning between it and a Task node, and to better represent its true meaning as a collection of activities.
- Branch node;
  - May have several (more than two) output streams.
  - The choice of output stream may be determined by another process participant, E.g. a project manager decides for a programmer which path to take and enforces that decision.
- Split node;
  - May have several (more than two) output streams.
  - Has been broadly redefined to allow for the decomposition of work, rather than just splitting work, thus reverting back to the original VPL meaning and eliminating the need for a Decompose node.
- Merge node;
  - May have several (more than two) input streams.
  - A Merge node allows for the recomposition of work, thus eliminating the need for a Recompose node.
- A Decompose node is now redundant.
- A Recompose node is now redundant.

4.1.2 The E/VPL paradigm

The VPL notion of an object is replaced by a more coarse grained, general entity called an Artifact, which would typically be used to represent entities like program specifications, test plans, schedules, code modules, change request forms, etc.

E/VPL enaction involves the flow of artifacts through a process program (graph). E/VPL drives the process by displaying where the participant(s) is in relation to the whole process and where they can go next. E/VPL enaction is based on the user (or process participant) updating the system with regard to their current position in the process program, i.e. the user, not the system, allows the artifacts to move to the next node in the process, not the system. This more relaxed method of movement through a process is in line with the central theme of E/VPL which is process support, rather than rigid enforcement of a process.

4.2 E/VPL Architecture

The architecture of the E/VPL system (see Figure 4.2) is considerably different to that of VPL (see Figure 3.1). The motivation behind the design of the E/VPL was to create a modularised architecture, with clear boundaries between each of the core elements of the PCE, so that any individual component of the system could be replaced, updated or altered without any disruption to the system as a whole. This architecture also creates distinct lines of communications between the various components of the system, allowing (if necessary) this communication to be tapped by other system components or external tools.

With E/VPL, the complete process model, including all data, is encapsulated in a single process database, whereas VPL uses flat files. E/VPL also has a separate process definition, process enaction and tools system, i.e. each part of the system may be

viewed as a component. This architecture separates the process model from the tools which work with on the model.

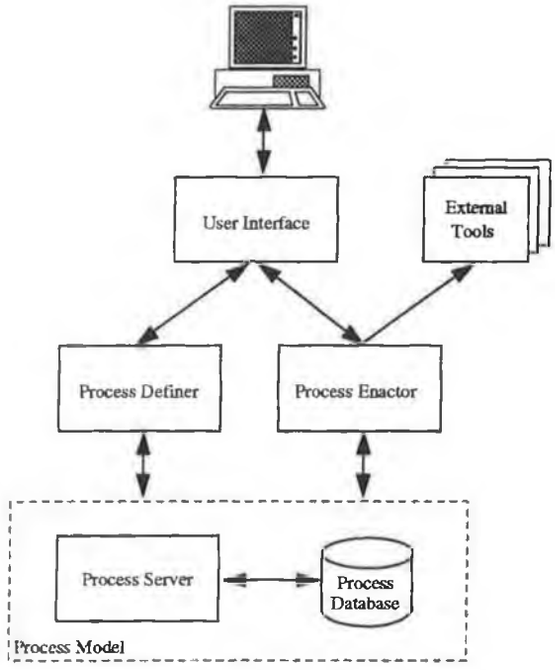

Figure 4.2 E/VPL architecture

In the following sections we will examine the various components of the E/VPL system in more detail.

4.2.1 The Process Database

The process model information is held in the process database, along with all enaction information. This information is collated with the E/VPL graph of the process by the process server to present the complete process model, or a partial version of it, to the other components of the system, i.e. the process definer, enactor or external tools via the enactor. The process database is the most important part of the system, as it will hold all information related to the process, yet little work has been done in the area of specifying database support for process support systems.

The notion of a project database is well understood, though rarely implemented in software development environments. The project database serves as a repository in which all of the information associated with a system or project is stored. A project database should store everything from requirements to source code and from user documentation to executable programs and program units, replicated as required to support multiple versions of each. Then, in principle, one can make a query against the database to determine the status of the project that it is supporting.

Traditional relational data models are record-orientated, and view the database as flat and simply structured, allowing the construction of larger objects through join operations on the base relations. A strong argument against the use of relational systems in environments with sophisticated data structures is that complicated objects are hard to map. A secondary argument is that users can attach no behaviour to their base entities. This is exactly one of the goals of object-orientated systems. Some work has been done in the area of mapping the requirements of PCEs to object-orientated databases [DP92], but this area still has to be further investigated.

With E/VPL the process database will hold all the process model information, with access to the database being made via a process server mechanism as described in the next section. The type of database chosen, relational or object-orientated, should not be of concern to the rest of the systems components, as their only view of the database will be provided by requests made to the process server. Consequently the process database may be altered without affecting the rest of the system.

4.2.2 The Process Server

The purpose of the process server is to service requests from other components of the system for information held in the database. The reason for having a process server, rather than allowing each component to send requests directly to the database, is to

provide a uniform mechanism for accessing the process database for all the components of the system and for external tools via the enactor.

The process server will receive requests to retrieve and update information from the other system components, convert these into queries (represented in a suitable language, such as SQL) and return the information to the component concerned in the appropriate format. By being the only interface between the database and the rest of the system, the process server will be responsible for all database maintenance, integrity and consistency checking and verifying the integrity of all database requests. Providing this layer between the database and the rest of the system means that the database is shielded from accidental damage by any individual component or external tool. It also enforces a strict authorisation mechanism, thus not allowing unwanted database access. The process server should ideally be viewed as a demon process, which is constantly running and servicing requests as necessary.

4.2.3 The Process Definer

The Process Definer is a tool exclusively for the process programmers use. It is used to define the process model, including the E/VPL graph in the first instance and also to alter aspects of the process model at a subsequent stage. The process programmer has the ability to alter the process model via the process definer if it is considered to be in an inconsistent state, or if an override mechanism is needed.

The reason for having the process definer as a separate component is to detach the specification of a process program from the execution of the program, a division that is not made in VPL. The process definer will provide a graphical means of defining a process, along with all the necessary elements to allow for capture of process related data. This information will be stored in the process database, via the process server. Apart from entering the process program, the process definer also allows for; entering artifact information, links between artifacts and activities, requirements that certain

activities may have, time constraints on activities, general project information and checking on project histories. The process definer also provides a by-pass mechanism. which may need to be invoked from time to time during the execution of a process program and a mechanism to return the process model to a consistent state after some possible error.

The process definer should be used exclusively by the nominated process programmer(s) to ensure consistency between the process program, updates, and alterations and to avoid unauthorised alteration of a process program.

4.2.4 The Process Enactor

The Process enactor provides the enaction (execution) engine for the process program. doing the main work of the PCE. The reason for separating the enactor and definer (unlike the VPL environment) is to show the distinction between creating a process and enacting it..

As would be expected, the process enactor provides the necessary mechanism to move the current state of the process model to another state, via the rules set down by the process programmer (using the process definer tool). The user (or process participant) is responsible for driving the process program. To enable this, the process enactor must represent the process model in its graphical form, making available all necessary process data and showing the user the possible path(s) forward from their current position in the process model. Thus the user is provided with all the information necessary to complete the current activity and then tell the system that the activity has been completed and choose from the next available activities.

Apart from simply registering movement from one state to another in the process program, the process enactor is also responsible for collecting process execution data. such as metrics (E.g. time taken by an activity versus time allowed) and recording the

process enaction history. The recording of a process history is very important, as it provides both a project log and a mechanism for undoing actions performed during execution. This undo mechanism is similar to a rollback in a traditional database using the logs, which revert it to a previous state.

4.3 Summary

In this Chapter, I have outlined the design issues surrounding process support systems in general and the main requirements of a PCE. From this basis and considering the arguments of Chapter 3, I have proposed enhancements and extensions to VPL. These enhancements have been outlined and a detailed view of the proposed system architecture given. The next logical step is to implement a software system based on E/VPL and apply a case study to it. This is the subject of Chapter 5.

CHAPTER 5 - IMPLEMENTING E/VPL

5. Introduction

Chapter 4 described E/VPL and its architecture. In this Chapter, I will describe the process of implementing a prototype E/VPL system. Firstly, I will introduce the design of the prototype, then go on to describe how the system was implemented. Finally I will describe a substantial example process program using E/VPL, in order to assess E/VPLs suitability as a process programming system.

5.1 Prototype Design

The prototype is designed around the component architecture of E/VPL (cf. section 4.2), for each component part, there is a corresponding system component. The motivation for this is, in part, to create a system of interchangeable software components and also to have work clearly divided between software components. The prototype has been designed to operate in Microsoft Windows in a networked personal computer environment.

The user interface is the central element of the system, which brings together the other system components. It is implemented as a set of menus allowing access to the process definer, process enactor and any external tools that have been linked into the system.

At the core of the system is the process database. This has been designed using the relational data model. The choice of data model was purely arbitrary, as there is no consensus of opinion in the research community as to which data model is best suited (cf. section 4.2.1). The database holds all the information on the process; including all artifacts, users, roles and metrics data. It also holds information about the actual

process diagrams, from which a graphical description can be rebuilt when necessary. The database is composed of over 18 tables, each with a primary key and a relationship (link) to other tables, which allows "joins" to build larger complex data structures.

The process server is the interface between the process database and the rest of the system. It receives standard SQL queries from the other system components and passes them on to the process database, creating a snapshot of the data required. It is the job of the process server to implement record locking, to ensure the integrity of the database after any updates and to verify the authorisation of any user / role that is updating the database.

The process definer and enactor components are accessed from the menu system of the user interface. They both share the ability to show the graphical model and other textual data about the system. It is the process definer's job to allow the user to draw a process graph and to validate it, before entering that information into the process database. The process enactor then use the process graph and other information to execute the process in real time, in accordance with the conditions described in the process model.

5.2 Implementing the Prototype

This prototype system has been implemented using Microsoft Visual Basic (version 3 professional) and Microsoft Access (version 1.1). The language was chosen as it provides all the necessary features to create a fast prototype system in Windows, with the database, because Microsoft Access provides all the facilities required to implement the process database and is recognised as the leading Windows database system.

The prototype contains over 10 KLOC, and is made up of 16 windows forms (screens) and 3 code modules. The main entry point into the system is a set of standard Windows menus, each of which calls an appropriate Windows form to handle the

desired action. The database and programs reside on a Novell network, with access being controlled by a user name and password system, with each user being assigned a role by the process programmer. Each user must first login to the system and is then presented with the menu system, or a restricted menu system, depending on their role.

The process server is passed a standard SQL query by the definer or enactor and uses this to create a dynaset (dynamic set) of the data required. This dynaset can then be queried by the definer or enactor in much the same way as a traditional database snapshot. In some cases the Visual Basic data control has been used for ease of programming, when data from a single table is being displayed but not updated. The process server must also check the authorisation level of the user to see whether the user is allowed to access the information (each user / role combination an authorisation level, which is used to limit activities).

5.3 The Prototype at work

In this section I will describe a typical session with the E/VPL system from the process programmers point of view. The purpose of this is to give a feel for how the system works from a users perspective. I shall describe the basic steps in creating a process program, including entering all sundry process data (artifacts, roles, etc.), i.e. how to use the Process Definer.

The E/VPL system is presented as an icon on the users Windows desktop, which they can double-click to activate the system. The user must then log into the system (see Figure 5.1), with the login information being used to look up the users role information in the process database and set the appropriate authorisation level. In this case, the process programmer will have full unrestricted access to the system.

Once authorisation has been verified the process programmer is presented with the main menu (see Figure 5.2). The process programmer will normally start by defining

the process participants (users), their roles and levels of authorisation within the system. Next, artifacts are defined by entering their names, where they are stored (paper file, electronic file, etc.) and the level of authorisation that a user must have to access them.

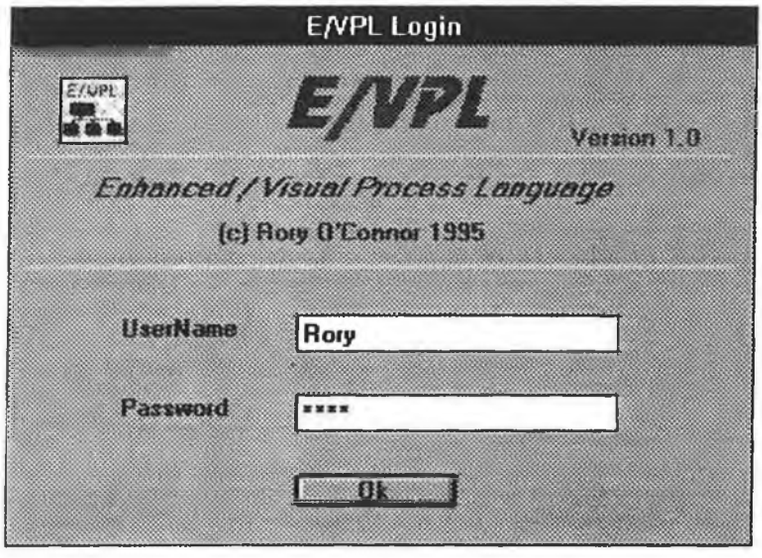

Figure 5.1 E/VPL login screen

The process programmer will then define the graphical process model using the E/VPL graphical formalism. This is done by selecting the definer drawing tool and the process programmer is then presented with a drawing page (see Figure 5.3), much like any other Windows graphics program. The drawing screen contains a toolbar from which the process programmer selects the node to be drawn and drags it onto the drawing page. The node is then validated to ensure it is being drawn in an allowable place (i.e. not obscuring any other object). The process programmer will normally enter the name of the node, a brief description of it, the estimated starting and finishing time (during execution) of the node and also the authorisation level required to access that node.

Figure 5.2 E/VPL main menu

This procedure is repeated until all the nodes are in place, at which point the process programmer draws the arcs between objects by simply clicking the right mouse button over the nodes to be connected and releasing the mouse. The arc is then validated to ensure it corresponds to correct E/VPL graph rules and that the diagram as a whole is valid.

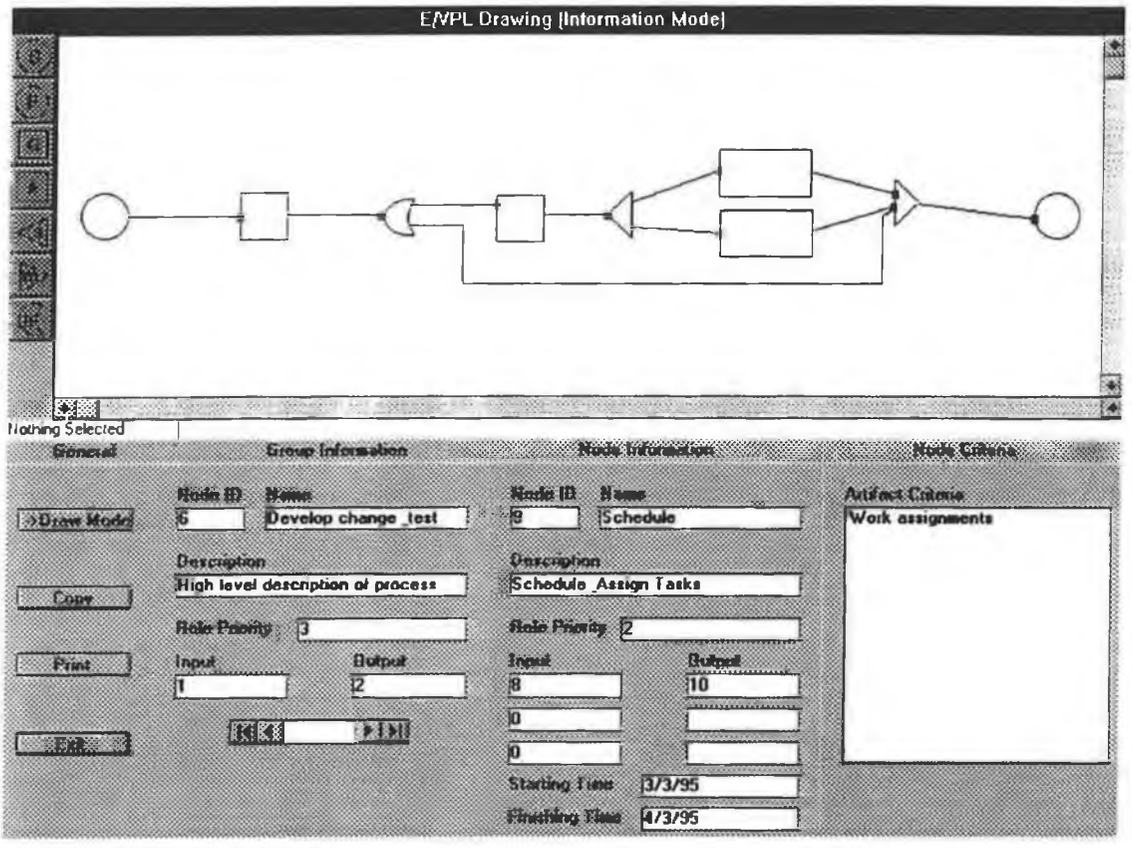

Figure 5.3 E/VPL process drawing screen

After constructing the complete process program, the process programmer must enter other information about the program. Normally the criteria that link nodes and artifacts are entered next. This is done by evoking a criteria tool within the process definer (see Figure 5.4) and defining the links and types of links between the nodes and the process artifacts. Also, information about the project as a whole is recorded, such as the estimated starting and finishing time of the project, the project name, etc. This is done to help build up a project history log.

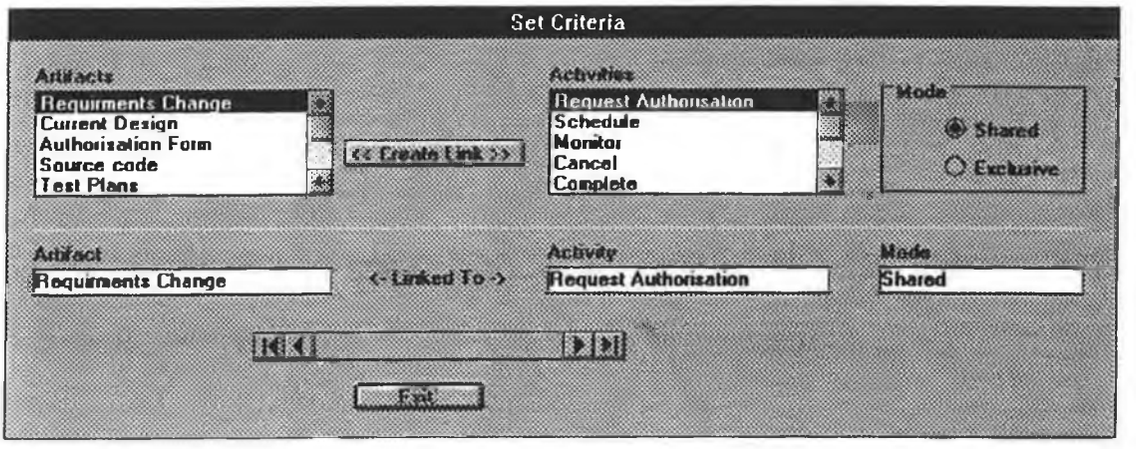

Figure 5.4 E/VPL artifact criteria screen

It is the job of the process programmer to ensure that the process program is a correct interpretation of the process being followed. Ideally this will already have been decided at some preliminary stage in the process description, i.e. the project personnel will have defined the project "on paper" using the E/VPL notation and simply transfer this model to the system, adding extra data as required. At this point the process program is ready for execution or simulation.

The Process Enactor operates in the same manner as the Process Definer. The user will login into the E/VPL system to identify themselves and then choose from the available options. The main enaction screen is similar to the main process drawing screen, with the currently active activity group being displayed. The next section shows the complete defination of a large process modelling problem that has been enacted using E/VPL.

5.4 The ISPW-7 problem

As part of the Seventh International Software Process Workshop (ISPW-7), participants were invited to solve a process modelling example [HK91]. This example problem was designed to aid understanding and comparing various approaches to software process modelling. Secondary goals included communicating the diversity of

aspects of processes encountered in modelling real-world software processes and providing impetus for efforts to extend the proposed approaches to cope effectively with these process issues. In the following sections I will describe the E/VPL approach to implementing a solution for ISPW-7.

5.5 The E/VPL approach to ISPW-7

The core problem is scoped as a relatively confined portion of a software change process. It focuses on the design, coding, unit testing and management of a localised change to a software system. This is prompted by a change in requirements and can be thought of as occurring either late in the development phase or during the maintenance phase of the life-cycle. For a full description of the problem refer to the ISPW-7 proceedings [HK91].

5.5.1 E/VPL Solution

E/VPL models the example problem in an intuitive and structured manner. As the example problem is broken into components sections, it is natural to gather the components with a common conceptual objective into an E/VPL activity group, with each component at its most refined stage being represented as an activity. Thus, the example problem as a whole can be represented as a single activity group called *develop change and test unit.* The example definition refers to this as the "highest-level abstraction" of the problem. As the problem is further refined, we can represent ever increasing levels of complexity using parallel activity groups, branches and eventually (atomic) activities.

For every node type there is a narrative description, including all information about the node, its location, function and flows to and from it. In the following sections each

activity group in the process model is represented and refined until all its activities have been represented.

5.5.2 Enaction of the ISPW-7 problem

The process program is enacted by allowing artifacts to move from the start node of the initial activity group into subsequent nodes. The major decision at this point is to decide on what artifacts should be available in the system. This decision is not as critical as it might seem, as artifacts may be created at any time during the enaction of the process. The reason for this is that certain artifacts may not exist at the beginning of the process model, e.g. a code file artifact will presumably only be created after design activities, so it cannot be available before design has been completed.

The decision as to what level of detail the artifacts should contain (coarse grained or fined grained) is up to the process programmer and depends on the particular process being modelled. The E/VPL system allows for artifacts to be decomposed into finer levels of detail and/or broken into component artifacts during the execution of a process program. This feature is necessary, as often a system specification may initially be small, but may at a later date be separated into several specifications to be worked on by more than one development team at the same time.

Artifact name	Description	Stored
Requirements change	Document detailing change required	File
Authorisation form	CCB's authorisation to carry out software change	Paper
Current design	Software design document	File/Paper
Source code	Source code undergoing changes	File
Test Plans	Plans for testing software after change	File
Schedule	Schedule of dates for major deliverables	File
Work assignments	Schedule of persons assigned to activities	File

Table 5.1 List of E/VPL artifacts for ISPW-7

Table 5.1 describes a set of artifacts that could be available at the start of this example. it also gives the name of the artifact, a short description of it and whether it is stored manually (paper), computer (file) or both.

Also available is a set of roles, which are assigned to the participants in the process model. Table 5.2 lists the roles necessary for the enaction of this process model and the number of participants needed to fill those roles. It should be noted that one person may occupy more than one role.

Role	Number required
Process Programmer	One
Project Manager	One
Design Engineer	Two
Quality Assurance Engineer	Two

Table 5.2 List of roles in system

5.5.3 ISPW-7 Process Model

The ISPW-7 problem is represented as a single activity group called "Root" (cf. section 5.3.3.1). This activity group is further refined to show all atomic activities in the process model. Each of the activity groups is described in the following sections.

5.5.3.1 Root activity group

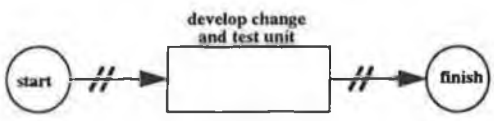

Figure 5.5 Root activity group

Activity group: Root

Description: This is a high level description of the process, it will be further decomposed into lower levels. It is presumed that this process resides in a larger process (indicated by // markers on either side of the activity group).

5.5.3.2 Develop Change and Test Unit activity group

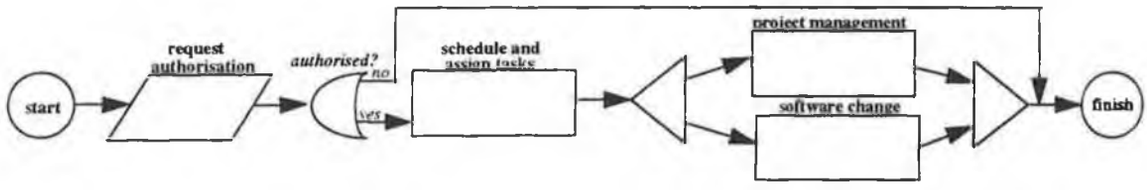

Figure 5.6 Develop Change and Test Unit activity group

Activity group: Develop change and test unit

Description: This activity group can only be processed if the Configuration Control Board (CCB) has authorised it. If authorisation has not been given the activity group will finish; otherwise, artifacts enter *the schedule and assign tasks* activity group, where all personnel and resources are assigned to the various activities. The next two activity groups occur in parallel with identical copies of all artifacts available to the two parallel flows (the management of the project and the software change itself). This activity group ends when all actions associated with the change have been carried out, or if authorisation was not given.

Activity: Request authorisation

Roles: Project manager

Description: The project manager requests authorisation from the CCB to make the change, with permission being either granted or refused.

Branch: Authorised

Edges: Yes, No

Description: If yes, then proceed to next node (*schedule and assign tasks* activity group), otherwise jump to finish.

5.5.3.3 Schedule and Assign Tasks activity group

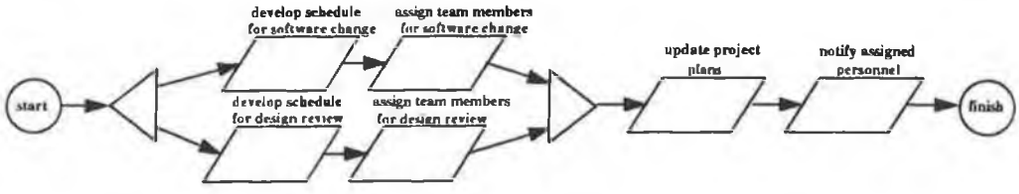

Figure 5.9 Schedule and Assign Tasks activity group

Activity Group: Schedule and Assign Tasks

Description: This represents the project management step of the process. It involves developing a schedule for the work and assigning individual activities to specific team members. The design review is also scheduled and participants are assigned to it. Both sets of activities are completed in parallel, after which all artifacts are merged, the project plans updated to reflect the schedule and personnel notified of their activity assignments.

Activity: Develop schedule for software change

Roles: Project manager

Description: Develop a schedule for the work to be undertaken for the software change, which is based on the standard process already in use by the organisation. Resources must be assigned to activities and an estimation of the schedule required for those resources to carry out the activities.

Activity: Assign team members for software change

Roles: Project manager

Description: Members of the project team, specifically a design engineer and a quality assurance engineer must be assigned to specific activities. The

preparation for this will have been done in the *develop schedule for software change* activity.

Activity: Develop schedule for design review

Roles: Project manager

Description: Develop a schedule for the work to be undertaken for the design review which is based on the process already in use by the organisation. Resources must be assigned to activities and an estimation of the schedule required for those resources to carry out the activities.

Activity: Assign team members for design review

Roles: Project manager

Description: Members of the project team, specifically a design engineer, quality assurance engineer and two other software engineers must be assigned to specific activities. The preparation for this will have been done in the *develop schedule for design review* activity.

Activity: Update project plans

Roles: Project manager

Description: The project plans must be updated to reflect the changes made in the previous activities.

Activity: Notify assigned personnel

Roles: Project manager

Description: All members of the project team who have been assigned activities must be notified of them and all scheduled dates. This is to be done either verbally or by e-mail.

5.5.3.4 Project Management activity group

Activity group: Project Management

Description: Artifacts pass through the initial start node and enter the *monitor progress* activity. This activity will live for the duration of the project and involves the project manager monitoring the progress and status of the work. If work proceeds according to plan, no action is taken until the project is completed. If however, deviations from the plan occur, they can result in re-scheduling of the project (i.e. re-execution of the *schedule and assign tasks* activity group) or the project manager can recommend to the CCB that the project be cancelled.

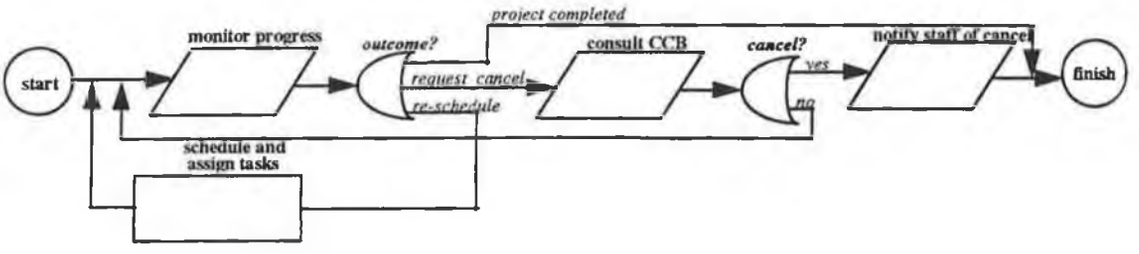

Figure 5.7 Project Management activity group

Activity: Monitor progress

Roles: Project manager

Description: The project manager monitors progress and status of the work. This is based on notification of completion of each step, as well as informal information. While work proceeds according to plan, no action is taken and no output is developed.

Activity: Consult CCB

Roles: Project manager

Description: If the project manager decides to request cancel, then this decision must be approved by the CCB.

Activity: Notify staff of cancel

Roles: Project manager

Description: Notify staff that the project has been cancelled.

Branch: Outcome

Edges: Project completed, Request cancel, Re-schedule

Description: If the project is completed then jump to finish, if request to cancel then proceed to consult CCB activity group, otherwise re-execute the *schedule and assign tasks* activity group.

Branch: Cancel

Edges: Yes, No

Description: If yes then proceed to notify staff of cancel activity group, otherwise loop back and re-execute *monitor progress* activity.

5.5.3.5 Software Change activity group

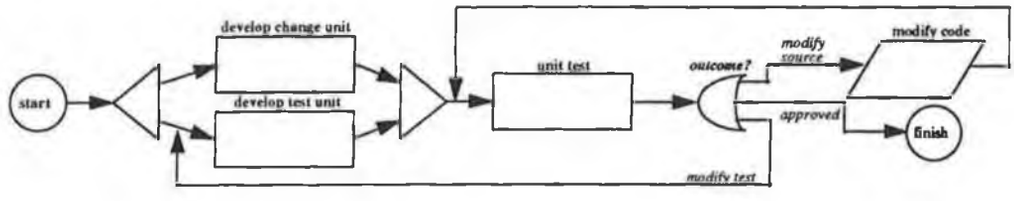

Figure 5.8 Software Change activity group

Activity group: Software change

Description: This shows the structure of the software change process. The activity groups *develop change unit* and *develop test* unit are carried out in parallel and then both merged for the *unit test* activity group. If all tests are successfully completed then the unit has been approved and the activity group terminates. If this is not the case, control branches in either of two other directions; the source code needs to be modified further, or the test unit needs to be modified further.

Branch: Outcome

Edges: Approved, Modify source, Modify test

Description: If approved then proceed to finish, if modify source then execute *modify code* activity, otherwise loop back and re-execute *develop test unit* activity group. (Note: approved decision is arrived at in the previous node, *unit test* activity group, i.e. "yes" in acceptable branch node. The other two decisions are also made in the previous node, in the analyse activity).

Activity: Modify Code

Roles: Design engineer

Description: This activity group involves the implementation of the design changes into code and compilation of the modified code into object code. This activity may also be based upon feedback from testing indicating that additional source code modifications are required.

5.5.3.6 Develop Change Unit activity group

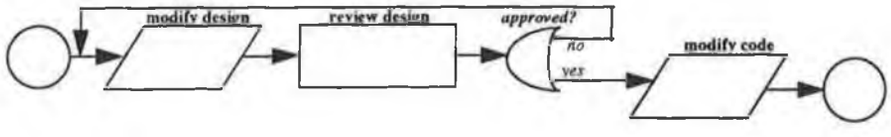

Figure 5.10 Develop Change Unit activity group

Activity Group: Develop Change Unit

Description: This activity group involves the modification and review of the design. The first activity to be carried out is the modification of the design, after which the design review is conducted. Control then branches in one of two directions; if approval is given the code modification may begin (execute the *modify code* activity), otherwise the design must be further modified (iterate this activity group).

Activity: Modify design

Roles: Design engineer

Description: This activity involves the modification of the design for the code unit affected by the requirements change. This step may be based upon feedback from the design review.

Activity: Modify Code

Described in Software Change activity group (cf. section 5.5.3.5).

Branch: Approved

Edges: Yes, No

Description: If yes then proceed to modify code activity, otherwise loop back to modify design activity group. (Note: "yes" and "no" are used to drive the outcome branch node of the review design activity group. i.e. yes = unconditional approval, no = minor / major changes. This is an arbitrary mapping).

5.5.3.7 Develop Test Unit activity group

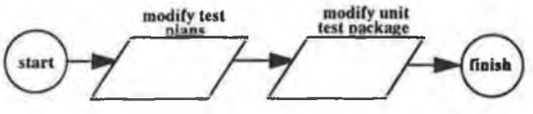

Figure 5.11 Develop Test Unit activity group

Activity Group: Develop Test Unit

Description: This activity group involves the modification of the test plans and of the actual unit test package for the affected code unit. The first activity involves the modification of the test plans and objectives to include testing of capabilities related to the requirements change prompting this software modification. These test plans are analogous to software designs. They identify the functionality and capabilities to be tested, and

the approach to be taken. The unit test package activity is modified in accordance with the changes made to the test plans and objectives. Subsequent iterations of this step may be based upon feedback from testing.

Activity: Modify test plans

Roles: QA engineer

Description: This is the modification of the test plans and objectives to include testing of capabilities related to the software requirements change. They identify the functionality and capabilities to be tested and the approach to be taken. The specifics of actual test data are handled in the *modify unit* test package activity.

Activity: Modify unit test package

Roles: QA engineer

Description: This activity involves the modification of the actual unit test package for the affected code unit, in accordance with the modifications made to the test plans and objectives. A new version of the unit test package is created when this activity is finished. Subsequent iterations of this activity may be based on feedback from testing.

5.5.3.8 Review Design activity group

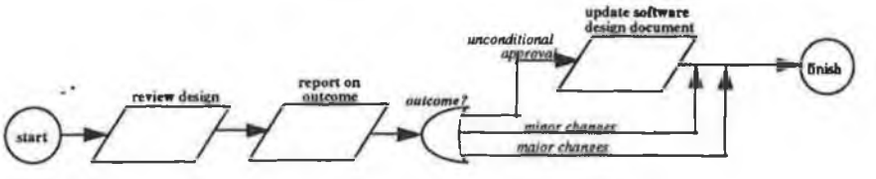

Figure 5.12 Review Design activity group

Activity Group: Review Design

Description: This activity group involves the formal review of the modified design. It is conducted by a team including the design engineer (who produces the design modifications). After the review is completed and a decision is made as to the design, there are three possible outcomes of the review:

1. Unconditional approval: the design is totally approved and is incorporated into the software design document; appropriate feedback is given.
2. Minor changes recommended: minor changes to the design are required and feedback is provided to the designer, with the re-review expected to be perfunctory.
3. Major changes recommended: major changes to the design are required and feedback is provided to the designer.

It is at this point that approval for *develop change unit* activity group is granted or not. This information is included in the feedback activity.

Activity: Review design

Roles: Design engineer and two software engineers

Description: This activity is the formal review of the modified design. There are three alternative outcomes of the review; unconditional approval, minor changes recommended, or major changes recommended.

Activity: Report on outcome

Roles: Design engineer and two software engineers

Description: This activity follows directly from the *review design* activity. Certain information is recorded about the design review. In particular, the number of defects identified and the aggregate effort of the review team in preparing for and conducting the review are reported to the project manager. The outcome agreed from the previous activity group (*review design*) is also reported to the project manager by e-mail.

Activity: Update software design document

Roles: Design engineer and two software engineers

Description: This activity will happen only if the outcome from the *review design* activity is unconditional approval. If this is the case then the approved design is incorporated into the software design document.

Branch: Outcome

Edges: Unconditional approval, Minor changes, Major changes

Description: If unconditional approval, proceed to update software design document activity group, otherwise jump to finish.

5.5.3.9 Unit Test activity group

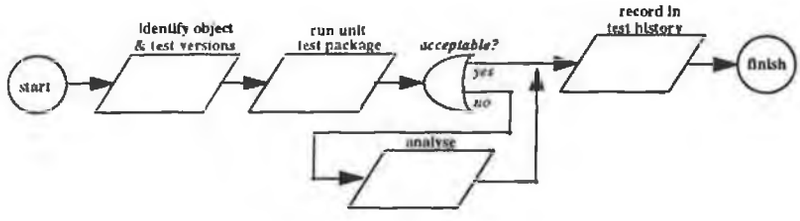

Figure 5.13 Unit Test activity group

Activity Group: Unit Test

Description: This activity group involves the application of the unit test package to the modified code and analysis of the results. The first activity is the identification of the object code and test unit versions, prior to a run of the entire test package. Although unit tests are primarily functional, an automated coverage analyser is employed to deterimne that adequate test coverage of the units code has been achieved; a 90% threshold has been established as acceptable. If all tests are successfully completed and 90% coverage attained, then the unit test passes and the results of the tests are recorded in the test history for the software unit. Otherwise the design and QA engineers jointly analyse the test results and

determine appropriate actions; the source code needs to be modified, or the unit tests need to be modified and further tests carried out, before recording this information in the test history. It is this decision which determines the branching in the *software change* activity group (i.e. approved, modify source, or modify test).

Activity: Identify object and test versions

Roles: Design engineer and QA engineer

Description: Prior to the running of a unit test (*run test unit package* activity), the version of the unit test package and that of the object code to be tested must be identified.

Activity: Run unit test package

Roles: Design engineer and QA engineer

Description: This activity is the actual application of the unit test package on the modified code. The entire test package is run before any analysis of further action is taken. Although the unit tests are primarily functional, an automated coverage analyser is employed to determine that adequate test coverage of the unit's code has been achieved; a 90% threshold has been established as acceptable. If all tests are successfully completed, and a 90% coverage attained, then the unit test has successfully passed.

Activity: Analyse

Roles: Design engineer and QA engineer

Description: If the unit test was not deemed to be successful in the *run unit test package* activity, then the design and QA engineers jointly analyse the test results and determine appropriate action. The only possible action that can be decided are; modify the source code further, or modify the unit test package further. This decision is used to drive the *outcome* brancg in the *Software Change* activity group.

Activity: Record in test history

Roles: Design engineer and QA engineer

Description: The results of both the *run unit test package* and *analyse* activities are recorded in the test history for the object code. This information includes the version information from the *identify object and test versions* activity, along with a timestamp of the test execution. indication of tests failed and the coverage attained.

Branch: Acceptable

Edges: Yes, No

Description: If yes, proceed to *record in test history* activity, otherwise execute *analyse* activity group.

5.6 Summary

In this Chapter we have seen how E/VPL was implemented and how the system operates for a typical session with a process programmer. We have also seen how E/VPL can be applied to a "normal" software development process. The next logical step at this stage is to assess E/VPLs solution and compare it to other solutions proposed by researchers in the process community. This and a discussion of the future of E/VPL will be the subject of Chapter 6.

6. Introduction

The research reported in this thesis has centred around the development of an automated system to support the description and enaction of software process models. The proposed E/VPL system is based on the VPL environment with enhancements to cope with some of the problems identified with VPL and encompass lessons learned from reviewing other process modelling research. The system has been prototyped and a case study has been performed. In this Chapter, I will examine issues which have yet to be answered in this field of research and evaluate E/VPL as a PCE.

6.1 Research issues

The immaturity of the process modelling field is manifest in the still high level of semantic ambiguity and in the lack of consolidation in process representation languages. It is a young area with many open issues, especially in the following areas [ABGM92];

- Representation and interaction with humans.
- Mechanisms to manage failure and unforeseen events.
- Mechanisms to guarantee that modification of the process "on the fly" is done in a disciplined and controlled way.
- Integration between process and data modelling.

Although software process engineering has established its position within the software engineering community, still no agreed-upon common terminology exists [Sta93];

- What is the basic vocabulary needed to model software development processes.
- A versatile and flexible meta-model should define the process of process modelling, stating precisely the context of the basic terminology.
- There is no clear distinction between a goal and the actions which satisfy that goal. In industrial practice the actions leading to a goal are often not specified.
- Real processes are based upon human interactions like decisions, augmentations, discussions, etc. A terminology should support those concepts.

One remaining issue is that of environment support. There is, at present, little agreement on the standards for environment design and support. [NIST93] is an attempt at producing a standard framework for environment support, but it is aimed at project support environments in general and not specifically at PCEs. The PCE requires special consideration due to its difference from other technologies (cf. section 1.1), with particular regard to the human factors (cf. section 6.2). Within the area of environment support, there is scope for the creation of standards for implementing database support for PCEs and SEEs in general.

6.2 Human factors in process automation

So far, I have dealt primarily with the technical issues associated with automating software processes. However, implementing a PCE involves more than just addressing the technology. The success of adoption rests at least equally as heavily on personnel, organisational and cultural elements. Adopting a new technology is likely to meet with significant resistance. As with any new technology, some of this will result from the fact that people do not like change from their routine to new and uncertain ways. However, some will come from reasons unique to process automation. Such reasons are related to the controllable nature of the technology and the automated collection of personal productivity metrics. Typical reactions to these changes will include [Chr95];

- Now I don't talk to people, I only communicate through my computer.
- I don't want management to know every move I make.
- I don't want to be treated like a cog in a machine.
- I know these metrics are going to be used against me in my annual evaluation.
- I don't have the control over things that I used to.

Such comments reflect the natural fears of staff members. Process automation imposes behavioural changes that are unique to the technology. Most computer tools are passive, in the sense that they respond to commands from a human agent. Process automation is different in that it can request actions of the human. If management wishes to succeed, then it needs to create an environment of trust that can only be achieved through closely involving the people who will have to live within the system. E/VPL provides a novel approach to this problem, by keeping the familiar role of a passive tool. It does this by eliciting the required information from the human agents. thus making them feel more comfortable with the new technology.

6.3 Evaluating E/VPL

With E/VPL, I set out to create a PCE that was simple to use and understand, yet powerful enough to represent large complex software processes. I chose the VPL notation, as it provided a formalism that was simple to understand and robust enough to represent complex processes. The E/VPL environment was based loosely around that of VPL, but with necessary alterations to provide the features expected of a modern PCE. In the following sections I will consider E/VPL under headings of: notation, environment and usage.

6.3.1 The E/VPL notation

The VPL notation was broadly maintained, with some subtle changes in terminology to avoid misinterpretation of process diagrams. Because the VPL notation has been successfully field tested at the Aurora software development unit (cf. section 3.4) and the E/VPL notation is based on that of VPL, it demonstrates that E/VPL is based on a proven notation. The ISPW-7 problem also allows for evaluation of E/VPLs notation. as it was fully represented using E/VPL, proving its ability to represent a complex software process. The next stage would be to complete a field test of a "real-world" process using E/VPL and demonstrate it to a group of software developers. This would yield valuable information as to the opinion of software developers on the notation of E/VPL.

6.3.2 The E/VPL environment

E/VPL was prototyped using Visual Basic (version 3 professional) and Access (version 1.1) in Microsoft Windows on a Novell network. The prototype system was tested using the ISPW-7 problem. As the system was a prototype and had to be built in a short period of time, I set out to implement only the core elements of a PCE, to prove (or otherwise) the suitability of such a tool to implement E/VPL.

Unlike the comparison between notations that can easily be made using the ISPW-7 problem, no such standard basis exists for comparing environments, therefore only basic observations may be made. The E/VPL architecture is based on a simple component style system, which I believe allows for flexibility in implementing the system and for future expansion. Many other environments have been implemented on an ad-hoc basis, with emphasis only on the formalism and not the environment structure.

When using the E/VPL prototype, some observations may be made about PCEs in general and E/VPL itself. PCEs need to be simple to use, with easy and quick access methods to process representations. With E/VPL, access to process diagrams was through a menu system, which in practice proved to be cumbersome. A better mechanism may have been to provide a series of Windows desktop icons, which could be clicked upon to display process diagrams, the current activity, etc. The awkward nature of traditional menu systems are in my opinion not suitable for PCEs, as information is needed frequently and quickly, but must also be easy to access, without intruding on the users workspace.

The enaction mechanism in E/VPL allowed for users to update the system by telling it which activity they wish to perform next. This was implemented by displaying all possible activities and then allowing users to select one. In reality, this proved to be an difficult mechanism to work with and, as stated above, an easier access mechanism is needed.

To fully evaluate E/VPL, a large-scale prototype system is needed, with full multi-user capabilities and a comprehensive enaction engine. This system could then be used as a basis for a comprehensive field test of both the E/VPL notation and environment using a real-word process. The prototype I developed has provided valuable and useful information, which can be used as a basis for the creation of such a large-scale E/VPL system.

6.3.3 The ISPW-7 solution

In order to fully evaluate E/VPL a major test case was needed. However, due to pressure on this research a full field evaluation could not be completed, so the ISPW-7 problem was used as a case study. With solutions submitted by over 13 different researchers, it provided a standard base from which to compare solutions. Having completed an E/VPL solution to ISPW-7 and compared it to others, the former stands

out as communicating the process in a form that is easily understandable to the project participants. Many of the solutions provided can only be understood by experts in the area of process formalisms and not by ordinary software project personal, who are the end recipients of the process descriptions.

There is a price to be paid for the relative simplicity of E/VPL as a PCE. Some of the solutions provide complex mechanisms to represent the subtleties in the ISPW-7 problem. E/VPL does not currently cater for resource allocation, project scheduling and complex iteration of a process. These criticisms apart, I consider E/VPL to be a suitable language for both describing and enacting process programs, as it places few constraints on the process itself, or on tools and allows users the flexibility to develop creatively while controlling process shortcuts and hence minimising the possibility of later problems. The ISPW-7 problem only provides a standard base from which to compare E/VPL to other process programming systems. To fully evaluate E/VPL a "real-world" process must be completed. This would be the next logical step in the evaluation procedure.

6.4 The future of E/VPL

Process automation is an immature area with scope for further research in all facets of the technology. E/VPL has demonstrated that such technology is feasible and may be used to produce software products within the framework of a quality process environment. The next step in this research is to implement a large-scale prototype system to implement E/VPL and to complete a full field test of E/VPL on a real-word process. The feedback gained from such an experiment would provide and solid basis for elaborating this research.

E/VPL has scope for further work which can help to add to the body of knowledge within the process automation community. Particular areas include;

- Enhanced user interface and system access mechanisms.
- Extended process measurement collection (i.e. metrics, etc.).
- The development of a process library to support process reuse.
- Investigation into object-orientated database support.
- Integration of external tools and the applicability of frameworks for integration such as PCTE.
- The area of communication among process participants needs more attention. In E/VPL, communication is handled on an ad-hoc basis, with no formal organisation. It may be possible to extend the E/VPL notation to explicitly include communication among process participants.
- Resource allocations features should be included. This could be added by the provision of a tool which could map resources (as objects) to process participants.
- A project scheduling tool should be added, either an external tool or an enhancement to the process enactor.
- Experience from modelling "real" processes will also have an impact on the future direction of this research.

As process automation research provides further and better solutions and the technology gains commercial acceptance, there is little doubt in my mind that process automation will help to shape the future of software engineering.

REFERENCES

[ABCFK91]
R.Aylett, H.Beck, P.Chung, J.Fraser, J.Kingston, "Development Environments", In Software Engineer's Reference Book, John McDermind(Ed.), Butterworth-Heinmann. 1991.

[ABGM92]
P.Armenise, S.Bandinelli, C.Ghezze, A.Morzenti, "Software Process Representation Languages: Survey and Assessment", In Proceedings of the 4th Conference on Software Engineering and Knowledge Engineering, IEEE Computer Society Press. 1992.

[ABGM93]
P.Armenise, S.Bandinelli, C.Ghezze, A.Morzenti, "A Survey and Assessment of Software Process Representation Formalisms", International Journal of Software Engineering and Knowledge Engineering, December 1993.

[AO93]
S.Arbaoui, F.Oquendo, "Software Process Modelling: Where are we?", Slides presented at the 2nd International Conference on the Software Process, February 1993.

[Bou93]
M.Bourdon, "Building Process Models using Process WEAVER: A Progressive Approach", In Proceedings of the 8th International Software Process Workshop, IEEE Computer Society Press, 1993.

[BMcD92]
A.Brown, J.McDermid, "Learning from IPSE's Mistakes", IEEE Software, March 1992.

[Chr93]
A.M.Christe, "A Graphical Process Definition Language and its Application to a Maintenance Project", Information and Software Technology, June/July 1993.

[Chr95]
A.M.Christe, "Software Process Automation", Springer-Verlag, 1995.

[CKO92]
B.Curtin, M.Kellner, J.Over, "Process Modelling", Communications of the ACM. September 1992.

[CL93]
J-Y.Chen, C-P.Lai. "An Enactable Software Process Modelling Approach". Information and Software Technology, October 1993.

[Cus91]
M.A.Cusumano, "Japan's Software Factories", Oxford University Press, 1991.

[DG90]
W.Deiters, V.Gruhn, "Managing Software Processes in the Environment MELMAC". ACM SIGSOFT Software Engineering Notes, December 1990.

[DP92]
A.Delis, G.Panagopoulos. "Database Support for Software Engineering Environments", In Proceedings of 1992 IEEE Conference on Systems and Cybernetics. IEEE Computer Society Press, 1992.

[EG91]
W.Emmerich, V.Gruhn, "FUNSOFT Nets: A Petri Net based Software Process Modelling Language", In Proceedings of the 6th International Workshop in Software Specification and Design, IEEE Computer Society Press, 1991.

[Eij89]
P.Eijk, "The Design of a Simulator Tool", Formal Description Technique, North-Holland, 1989.

[EJS91]
W.Emmerich, G.Junkermann, W.Schafer, "MERLIN: Knowledge-Based Process Modelling", In Proceedings of the 1st European Workshop on Software Process Modelling, IEEE Computer Society Press, 1991.

[Fer93]
C.Fernström, "Process WEAVER: Adding Process Support to UNIX", In Proceedings of the 2nd International Conference on the Software Process, IEEE Computer Society Press, 1993.

[FH92]
P.Feiler, W.Humphrey, "Software Process Development and Enactment: Concepts and Definitions", SEI Technical Report CMU/SEI-92-TR-04, Carnegie Mellon University, September 1992.

[FKN94]
A.Finkelstein, J.Kramer, B.Nuseibeh, "Software Process Modelling and Technology", Research Studies Press, 1994.

[Gru93]
V.Gruhn, "Software Process Simulation in MELMAC", Systems Analysis and Modelling Simulation, vol. 2(2), 1993.

[HFB90]
L.Hubert, F.Fournier, B.Brasseur, "Eureka software factory: OPIUM an environment for software process modelling integrated with a project management tool", In Proceedings of the 6th International Software Process Workshop, IEEE Computer Society Press, 1990.

[HK91]
D.Heimbigner, M.Kellner, "Software Process Example for ISPW-7", 7th International Software Process Workshop - Call for Participation, August 1991.

[Hum89a]
W.Humphrey, "Managing the Software Process", Addison-Wesley, 1989.

[Hum89b]
W.Humphrey, "Software Process Modelling: Principles of Entity Process Models", SEI Technical Report CMU/SEI-89-TR-2, 1989.

[Hum90]
W.Humphrey, "People considerations in process models", In Proceedings of the 6th International Software Process Workshop, IEEE Computer Society Press, 1990.

[Hum95]
W.Humphrey, "A Discipline for Software Engineering", Addison-Wesley, 1995.

[ISO89]
ISO 8807, Information Processing Systems - OSI - LOTOS - a Formal Description Technique based on the Temporal Ordering of Observational Behaviour, 1989.

[Jac83]
M.A.Jackson, "Systems Development", Prentice-Hall, 1983.

[JC93]
M.L.Jaccheri, R.Conradi, "Techniques for Process Model Evolution in EPOS", IEEE Transactions on Software Engineering, December 1993.

[Kel91]
M.Kellner, "Software Process Modelling Support for Management Planning and Control", In Proceedings of the 1st International Conference on the Software Process. IEEE Computer Society Press, 1991.

[KFP88]
G.Kaiser, P.Feller, S.Popovich, "Intelligent Assistance for Software Development and Maintenance", IEEE Software, May 1988.

[KH87]
M.Kellner, G.Hansen, "Software Process Modelling: A Case Study", In Proceedings of the 22nd Annual Hawaii International Conference on System and Sciences, vol. 2 Software Track, IEEE Computer Society Press, 1987.

[KM93]
M.Keller, N.Madhavji, "A Comprehensive Process Model for Studying Software Process Papers.", In Proceedings of the 15th International Conference on Software Engineering, IEEE Computer Society Press, 1993.

[KTLAE92]
H.Krasner, J.Terrel, A.Linehan, P.Arnold, W.Ett, "Lessons Learned from a Software Process Modelling System", Communications of the ACM, September 1992.

[KTO93]
W.Kleppinger, D.Tamanaha, L.Osterweil, "A Framework for Understanding the Uses of Process Modeling Formalisms", Technical Report - University of California at Irvine, 1993.

[Lot93]
C.Lott, "Process Measurement and Support in SEE's", ACM SIGSOFT Software Engineering Notes, October 1993.

[MS90]
P.Mi, W.Scacchi, "A Knowledge Based Environment for Modelling and Simulating Software Engineering Processes". IEEE Transactions on Knowledge and Data Engineering, September 1990.

[MS93]
P.Mi, W.Scacchi, "Modelling, Integrating and Enacting Software Engineering Processes", In Proceedings of CASE 93, 1993.

[NF93]
B.Nuseibeh, A.Finkelstein, "ViewPoints: A Vehicle for Method and Tool Integration". In Proceedings of CASE 93, 1993.

[NFK93a]
B.Nuseibeh, A.Finkelstein, J.Kramer, "Fine-grain Process Modelling", In Proceedings of the 7th International Workshop on Software Specification and Design, IEEE Computer Society Press, 1993.

[NFK93b]
B.Nuseibeh, A.Finkelstein, J.Kramer, "Expressing the Relationships between Multiple Views in Requirements Specification", In Proceedings of the 15th International Conference on Software Engineering, IEEE Computer Society Press, 1993.

[NIST93]
"Reference Model for Project Support Environments (Version 2.0)", Technical Report NIST SP 500-213, National Institute of Standards, also published as; Technical Report CMU/SEI-93-TR-23, Software Engineering Institute, November 1993.

[Ost87]
L.Osterweil, "Software Processes are Software Too", In the Proceedings of the 9th International Conference on Software Engineering, IEEE Computer Society Press, 1987.

[OZG91]
F.Oquendo, J.Zucker, P.Griffiths, "The MASP Approach to Software Process Description Instantiation and Enactment", In Proceedings of the 1st European Workshop on Software Process Modelling, IEEE Computer Society Press, 1991.

[PR88]
M.Penedo, W.Riddle, "Guest Editor's Introduction to Software Engineering Environment Architectures", IEEE Transactions on Software Engineering, June 1988.

[PS92]
B.Peuschel, W.Schäfer, "Concepts and Implementation of a Rule-Based Engine", In Proceedings of the 14th International Conference on Software Engineering, IEEE Computer Press, 1992.

[Sib91]
S.Sibbald, "Visual Process Language: Foundation for a Second Generation Software Engineering Environment", M.Sc. Thesis, Royal Military College, Canada, 1991.

[SKS91]
M.Saeki, T.Kaneko, M.Sakamoto, "A Method for Software Process Modelling and Description using LOTOS", In Proceedings of the 1st International Conference on the Software Process, IEEE Computer Society Press, 1991.

[SSW92a]
T.Sheppard, S.Sibbald, C.Wortley, "A Visual Process Language", Communications of the ACM, April 1992.

[SSW92b]
T.Sheppard, S.Sibbald, C.Wortley, "Software Process Enaction with VPL", In Proceedings of CASE 92, 1992.

[Sta93]
G.Starke, "Urgent Research Issues in Software Process Engineering", ACM SIGSOFT Software Engineering Notes, October 1993.

[TBCO88]
R.Taylor, F.Belz, L.Clarke, L.Osterweil, et al, "Foundations for the Arcadia Environment Architecture". In Proceedings of the 3rd ACM Symposium in Software Development Environments, October 1988.

[Tre90]
P.Tremblay, "A Proposed Software Engineering Environment for Systems Life Cycle Management", M.Sc. Thesis, Royal Military College, Canada, 1990.

[Tul86]
C.Tully, "Software Process Models and Iteration", In Proceedings of the 3rd International Software Process Workshop, IEEE Computer Society Press, 1986.

[Win73]
T.Winograd, "Breaking the Complexity Barrier (again)", In the Proceedings of ACM SIGPlan-SIGIR Interface meeting on Programming Languages - Information Retrieval, ACM, 1973.

APPENDIX A

The prototype E/VPL system contains over 10 KLOC, and is made up of 16 windows forms (screens) and 3 code modules. The main entry point into the system is a set of standard Windows menus, each of which calls an appropriate Windows form to handle the desired action. The database and programs reside on a Novell network, with access being controlled by a user name and password system, with each user being assigned a role by the process programmer.

The following table lists all the Visual Basic forms used in the system and gives a brief description of what each form does:

Visual Basic Forms

About	Display about box
ViewArtifacts	Browse up/down artifacts list
ChangeArtifacts	Add/update/delete artifacts
Criteria	Set artifact/node criteria
Drawing	Main E/VPL drawing page and all related functions
GetPassword	Get & verify password from user
GetRole	Set role/node criteria
EnactHistory	View enaction history
Login	Login routine
Main	Main menu
NodeUpdate	Manual node updates
Passwords	Change password
ProjectHistory	Update/view project history details
Roles	Add/update/delete role information
Running	Main E/VPL enaction page and all related functions
EnactStart	Enaction start-up routines

The following table lists all the Visual Basic code modules use in the system and gives a brief description of what each module contains;

Visual Basic Code Modules

Constants	All constants declarations
Datacons	All constants declarations for database
Global	Global routines and declarations

The following tables lists all the Access database tables used in the system and describes their name, datatype and gives a brief description of their purpose;

ActiveArtifacts

ArtifactID	Number	ID number for a artifact
UserID	Number	ID number for User who is currently using artifact
Mode	Number	0=Exclusive, 1=Shared

Activity

NodeID	Number	Unique ID number for node
Input1	Number	ID Number of previous node
Input2	Number	ID Number of previous node
Input3	Number	ID Number of previous node
Output	Number	ID Number of next node
Description	Memo	Narrative description
Priority	Number	ID number of lowest role which can act upon activity
StartTime	DateTime	Estimated starting time of activity
FinishTime	DateTime	Estimated finishing time of activity

Activity Group

NodeID	Number	Unique ID number for node
Name	Text	Name of node
Input	Number	ID Number of previous node
Input	Number	ID Number of previous node
Input	Number	ID Number of previous node
Output	Number	ID Number of next node
GroupStart	Number	ID of start node inside this activity group
GroupFinish	Number	ID of finish node inside this activity group
Description	Memo	Narrative description
Priority	Number	ID no. of lowest role which can act upon activity group

AllNodes

NodeID	Number	Unique ID number for node
Type	Number	0=Start, 1=Finish, 2=Activity Group, 3=Activity, 4=Branch, 5=Split, 6=Merge
X	Number	X position on drawing sheet
Y	Number	Y position on drawing sheet
ActivityGroup	Number	ID number of host Activity Group

Artifact

ArtifactID	Number	Unique ID number for a artifact
Name	Text	Name of artifact
Description	Memo	Narrative description
Stored	Text	Where stored (Values: File, Paper, Other)
Priority	Number	ID number of lowest role which can act upon this artifact

Branch

NodeID	Number	Unique ID number for node
Name	Text	Name of node
Input1	Number	ID Number of previous node
Input2	Number	ID Number of previous node
Input3	Number	ID Number of previous node
Output1	Number	ID Number of next node (Edge 1)
Output2	Number	ID Number of next node (Edge 2)
Output3	Number	ID Number of next node (Edge 3)
Description	Memo	Narrative description
DecisionPoint	Number	ID of node where decision is arrived at (usually previous)
Priority	Number	ID number of lowest role which can make a decision

Finish

NodeID	Number	Unique ID number for node
Name	Text	Name of node
Input	Number	ID Number of previous node
Description	Memo	Narrative description

Merge

NodeID	Number	Unique ID number for node
Name	Text	Name of node
Input1	Number	ID Number of previous node (Edge 1)
Input2	Number	ID Number of previous node (Edge 2)
Input3	Number	ID Number of previous node (Edge 3)
Output	Number	ID Number of next node
Description	Memo	Narrative description
Priority	Number	ID number of lowest role which can act upon this merge

Message

MessageID	Number	Unique ID number for a message
UserID	Number	ID of user to receive message
Subject	Text	Subject heading for message
Narrative	Memo	Text of message

NodeArtifactCriteria

NodeID	Number	ID number for a node
ArtifactID	Number	ID number for a artifact
Mode	Number	0=Exclusive, 1=Shared

NodeHistory

HistoryID	Number	Unique ID for a history
NodeVisited	Number	ID of node visited
EnterTime	DateTime	Entry time of user into node
ExitTime	DateTime	Exit time of user out of node
ActivityType	Number	Type of activity
UserID	Number	ID of user doing action

Numbers

DummyID	Number	Key for table (not necessary)
NextNodeID	Number	Next unique node ID number in series
NextArtifactID	Number	Next unique artifact ID number in series
NextUserID	Number	Next unique user ID number in series
NextRoleID	Number	Next unique role ID number in series
NextMessageID	Number	Next unique message ID number in series
NextHistoryID	Number	Next unique history ID number in series

ProjectHistory

DummyID	Number	Key for table (not necessary)
EstimatedStart	DateTime	Estimated starting time of project
ActualStart	DateTime	Actual starting time of project
EstimatedFinish	DateTime	Estimated finishing time of project
ActualFinish	DateTime	Actual finishing time of project
Name	Text	Project Name

Split

NodeID	Number	Unique ID number for node
Name	Text	Name of node
Input1	Number	ID Number of previous node
Input2	Number	ID Number of previous node
Input3	Number	ID Number of previous node
Output1	Number	ID Number of next node (Edge 1)
Output2	Number	ID Number of next node (Edge 2)
Description	Memo	Narrative description
Priority	Number	ID number of lowest role which can act upon this split

Start

NodeID	Number	Unique ID number for node
Name	Text	Name of node
Output	Number	ID Number of next node
Description	Memo	Narrative description

UserActiveNodes

NodeID	Number	Node ID number for currently active node
UserID	Number	User ID for user in Node
Host	Number	ID number of host activity group

UserRoles

RoleID	Number	Unique ID number for a role
Name	Text	Name for role
Priority	Number	Security level of a role, from 0 to N
Description	Text	Narrative description

Users

UserID	Number	Unique ID number for user
Username	Text	Username
Password	Text	User definable password
RoleID	Number	Role ID for user