THE UNIVERSITY OF NEWCASTLE UPON TYNE

SCHOOL OF COMPUTING SCIENCE

Supporting Software Processes

for

Distributed Software Engineering Teams

by

Kamal Zuhairi Zamli

Ph.D. Thesis

October 2003

Abstract

Software processes relate to the sequence of steps that must be carried out by humans to pursue the goals of software engineering. In order to have an accurate representation of what these steps actually are, software processes can be modelled using a process modeling language (PML). Some PMLs simply support the specification of the steps, while others enable the process to be executed (or enacted). When enacted, software processes can provide guidance, automation and enforcement of the software engineering practices that are embodied in the model.

Although there has been much fruitful research into PMLs, their adoption by industry has not been widespread. While the reasons for this lack of success may be many and varied, this thesis identified two areas in which PMLs may have been deficient: human dimension issues in terms of support for awareness and visualisation; and support for addressing management and resource issues that might arise dynamically when a process model is being enacted. In order to address some of these issues, a new visual PML called Virtual Reality Process Modelling Language (VRPML) has been developed and evaluated. Novel features have been introduced in VRPML to include support for the integration of a virtual environment, and dynamic creation and assignment of tasks and resources at the PML enactment level. VRPML serves as a research vehicle for addressing our main research hypothesis that a PML, which exploits a virtual environment, is useful to support software processes for distributed software engineering teams.

Acknowledgements/Dedication

First of all, I would like to dedicate my special thanks to my supervisor Professor Pete Lee, for his undaunted patience, guidance, and constant advice throughout my studies in Newcastle. He provided numerous constructive criticism and detailed comments. To say the least, without Pete's encouragement and enthusiasm, I will probably would not have gone this far. Also, even though Pete is very busy, he took an enormous task of revising my thesis word by word. His efforts are greatly appreciated and will never be forgotten. Thanks again Pete.

Secondly, I would also like to thank all members of staffs in the School of Computing Science particularly Professor Cliff Jones, Dr Christina Gacek, and (former staff) Dr Nick Rossiter for giving useful advice especially during the regular thesis comittee meetings. To my internal examiner, Professor Santosh Shrivastava, and my external examiner, Professor Brian Warboys, thank you for a fruitful viva session. To Shirley Craig, thank you for your patience and efficiency in searching out many relevant references in this thesis. To Gerry Tomlinson, Iain Wood, Malcolm Green, and Tim Smith, thank you for being prompt when fixing occasional hardware and software problems.

Finally, I would like to thank my wife, Mardiah Hassan, and four kids – Ainin, Arif, Syahidah, and Zarifah – for being patience all along. I am sorry to have sometimes neglected all of you to pursue my dream. To my dad and mom, thank you for the prayers – this thesis is for both of you. To my mother-in-law, thank you for always remembering to send us packages full of cookies every Eid.

The work reported here would have not been possible without the grants from Universiti Sains Malaysia.

Table of Content

List of Figures

List of Tables

Chapter 1 – Introduction

Engineering as a discipline relates to the creative application of mathematical and scientific principles to devise and implement solutions to problems in our everyday lives in an economic and timely fashion. To provide a quality solution, it is not usually sufficient to focus only on the final product. Often, it is also necessary to consider the *processes* involve in producing that product. For example, consider an assembly of a car. From the customer's perspective, it is the final product that matters (i.e. a quality car). From an engineering perspective, such quality could not be achieved if some of the processes (e.g. assembly lines) are faulty. Although additional rework can fix the problems caused by the faulty assembly lines, this tends to raise the overall costs because it deals only with symptoms of the problem. In contrast, going to the cause of the problem and improving the process (e.g. the faulty assembly lines) avoids the introduction of quality defects in the first place and leads to better results with lower costs. As this example illustrates, it is through the processes that engineers can observe and improve quality, control productions costs and possibly reduce the time to market their products.

Similar analogies can be applied in the case of software engineering. To produce quality software, it is also necessary to place emphasis on the processes by which the software is produced. In software engineering, these processes are usually called software processes.

1.1 Overview of Software Processes

Software processes can be defined as sequences of steps that must be carried out by humans (e.g. software engineers), to pursue the goals of software engineering. There are many ways that can be used to define a particular software process. Perhaps the simplest way to define the software process is to use a natural language such as English. For example, one may describe a software process for the unit testing stage of software development as the following steps:

- Step 1: Check out the affected modules from the configuration management system.
- Step 2: Obtain the test cases from the project manager.
- Step 3: Perform the testing for all the test cases.
- Step 4: Produce the test report for the project manager.

Using a natural language for defining a software process is straightforward but exhibits a number of difficulties. In general, the description of software process using a natural language is often imprecise, ambiguous, inconsistent and open to user interpretation. Typically, such characteristics may lead to discrepancies in the software process undertaken by software engineers – for example, what is performed may not be what is required in the description of the software process. To eliminate such discrepancies, there is a need for a more precise way of specifying a software process.

In software engineering, such a need is translated into the use of modelling languages to specify a software process. In particular, software processes can be specified using a *process modelling language* (PML) and assisted by an environment called *Process Centred Software Engineering Environment* (PSEE). Through the use of a PML, software processes can be described in a precise way in terms of what a process comprises and how it is structured and organised. This can be instrumental in eliminating inconsistencies in the process specification.

As its name suggests, a PML is used to construct a form of model of the actual software development process. Such a model is often called *process model*, which is a representation of the actual software process excluding details which do not influence its relevant behaviour. As an illustration, Figure 1-1 depicts the process model for the common software process problem called the ISPW-6 problem [48] expressed using the PML developed in this research (the Virtual Reality Process Modelling Language – VRPML).

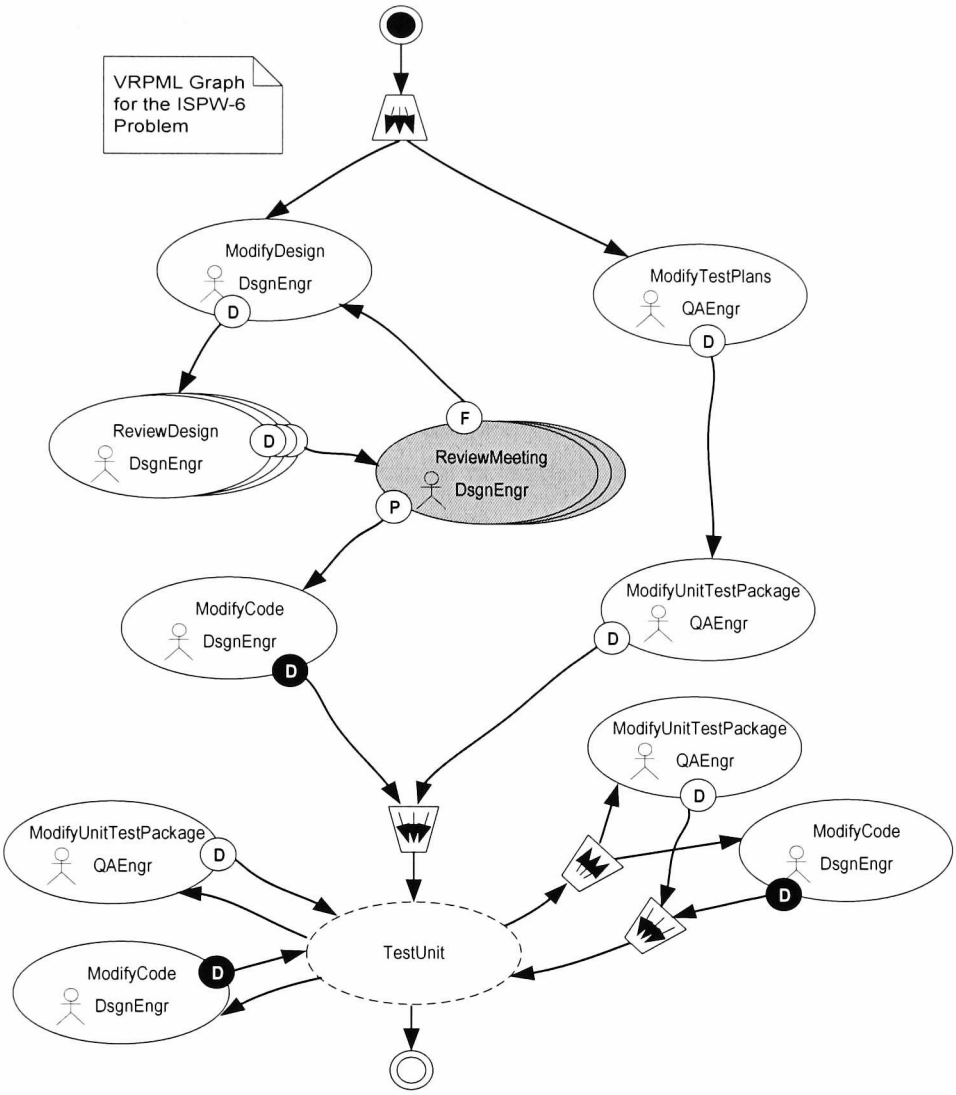


Figure 1-1 The ISPW-6 problem expressed in VRPML

Briefly, the ISPW-6 problem involves a software requirement change request occurring either towards the end of the development phase or during the maintenance and enhancement phase of the software lifecycle. When a software change request is received, the project manager assigns and schedules specific tasks to a number of participating software engineers. These tasks includes: Modify Design; Review Design; Modify Code; Modify Test Plans; Modify Unit Test Package; and Test Unit. Some tasks may be executed in parallel, while others have to be executed in a sequential manner. Not all tasks start at the same time (i.e. they require coordination). Although not completely shown in Figure 1-1, in each task, there are defined roles, tools, source files, task ordering constraints, and pre-

conditions and post-conditions which must be respected by the software engineers to complete the task. While the overall syntax and semantics of VRPML used in Figure 1-1 has not yet been fully explained (and is defined in detail in Chapter 4 of this thesis), the above model of the ISPW-6 problem provides some insights into the modelling of software processes and their degree of complexities.

In addition to being able to support the modelling of software processes, PMLs may also allow execution of the process models to support the activities of software engineers. The execution of such processes is usually termed *process enactment*. A significant number of PMLs surveyed in this research supports enactment, as will be discussed in Chapter 2. Enactment is very useful feature of a PML for the following reasons:

- It provides guidance through the steps to be taken. Such guidance is particularly useful for junior software engineers.
- It can enforce strict procedures and policies. Enforcement of strict procedures is sometimes important in cases such as developing critical systems where human lives depend on a piece of software. An example of such a system would be a car auto-cruise control system. In this case, the software development team in charge of developing such a system may require its defined steps to be followed precisely. For example, evolution of the software in such a system must be strictly controlled. *Ad hoc* changes must not be permitted because such changes may introduce bugs which may not be tested and accounted for. Such bugs could be dangerous especially if they affect the mechanism to control the speed of the car in auto-cruise.
- It permits the automation of tasks. In software engineering, there are many tasks which can benefit from automation. For example, although tasks such as compiling and linking source codes look simple, they can be painstakingly dull especially if the source codes are very large and involving multiple modules. Such mundane tasks, if automated, can relieve software engineers from tedious routine work (and reduce potential human errors), and consequently, improve software engineer's productivity.

Enactment raises two main issues concerning the dynamic nature of software processes and the human dimension. Because software processes are dynamic

entities, rarely can resources (i.e. in terms of engineers, artefacts and tools assignments) be completely planned ahead of time. Therefore, as will be seen in Chapter 2, there is a need to provide support for addressing management and resource issues that might arise dynamically when a PML is being enacted

Concerning the human dimension, enactment might also downplay the importance of human in a software development. Human are autonomous and creative and are unlikely to function well if treated like robots by completely prescribing their activities. Yet, human are also unpredictable and with degree of variability, as well as having a tendency to make mistakes, requiring some form of prescription to ensure necessary steps are faithfully followed (e.g. to define uniform software processes across different projects). It seems that such characteristics of humans are two sides of the same coin which warrants careful consideration. The human dimension issues are equally important in a PML and in its PSEE, although it is the PML that is main focus of this thesis.

As will be seen in the taxonomy of PMLs presented in Chapter 2, the support for the human dimension in present PMLs is still lacking, particularly in term of supporting software engineers working in geographically and temporally distributed locations. One way of enriching support for the human dimension is to incorporate issues from related research areas. In particular, cross-fertilisation from Computer Supported Cooperative Work (CSCW) – a field of study which puts emphasis on the nature of humans working together collaboratively to achieve a common goal as well as on the possibilities of technology to support and improve individual and group effectiveness, seem particularly appropriate [6, 32, 72]. Issues such as user awareness support, process awareness support, and process visualisation are seen as the key issues for improving the human dimension [72].

A subset of CSCW, Collaborative Virtual Environments (CVE), is of particular interests in this thesis. CVE advocates the use of a virtual environment to support collaborative working. By exploiting a virtual environment, a CVE is capable of providing a level of awareness, in terms of the other users involved and the processes being undertaken, which closely resembles the concept of awareness in the real world. This thesis looks into incorporating some of the human dimension

issues addressed by the CVE into the design and enactment of a PML, in particular, by exploiting a virtual environment.

The prospect of employing visual notations for a PML is also attractive to address the human dimension (e.g. in terms of simplifying the task of creating and comprehending a process model). In fact, a number of PMLs have already employed visual notations, as will be seen in the survey of PMLs in Chapter 2. The use of visual notations in a language, often termed a visual programming language, is not new. The basic idea behind a visual programming language is that computer graphics (e.g. graphs consisting of icons, nodes, and arcs) are used instead of a textual representation. The central argument for a visual programming language is based on an observation that picture is better than text (i.e. a picture is worth a thousand words). Such an observation can be supported by considering the ISPW-6 model expressed in VRPML in Figure 1-1. It can be seen that many aspects of the ISPW-6 model in Figure 1-1 should be intuitively obvious to the viewer even without knowing the full syntax and semantics of VRPML.

While a visual programming language may not be able to provide a silver bullet to simplify all aspects of engineering software, a carefully chosen level of abstractions (e.g. by working at the same level of abstraction as the problem domain) coupled with easy to understand visual notations may help alleviate the low-level complexities offered by the textual counterpart. Ideally, this makes a visual programming language more accessible to non specialist programmers. Because of such an alluring prospect, visual programming aspects also form an important part of this research to address the human dimension.

As discussed in preceding paragraphs, this work draws upon findings from research into CSCW and its subset CVE, and visual programming languages. This is especially important as generally for a piece of research to be successful, it must not concentrate solely on one key area, but must draw upon related research areas to gain depth and context. In line with such a notion, the hypothesis which suggests that a visual PML which exploits a virtual environment is useful to support software processes for distributed software engineering teams, is the main topic of this research.

1.2 Thesis Aims and Objectives

The main aim of this research is to produce and evaluate a new visual and enactable PML, VRPML, which addresses some of the perceived deficiencies in other PMLs. The main objectives of the work undertaken were:

- To identify salient characteristics and deficiencies in existing PMLs. Although research into software processes has been conducted for more than a decade, their usage has not been widespread. Thus, it is appropriate to identify the salient characteristics of existing PMLs, and to identify potential areas of omission, the rectification of which could improve the applicability of PMLs.
- To offer an evaluation of the existing PMLs using a common assessment grid. This provides a road map of the state of the art in the field.
- To build a prototype runtime support system to evaluate the feasibility, the usefulness, and the problems arising from the approach taken in this research. This provides insight into the true value of VRPML.

1.3 Thesis Outline

The remainder of this thesis is organised into eight chapters as follows.

Chapter 2 presents a classification and an analysis of the main characteristics of existing PMLs – forming a taxonomy for PMLs. Five main PML support areas are discussed: modelling support; enactment support; evolution support; evaluation support; and human dimension support. Using the taxonomy, a survey of PML is provided. Towards the end of Chapter 2, an analysis of PMLs is presented which provides the requirements and justification for the development of VRPML. In particular, the analysis highlights new areas of research relating to the support for human dimension issues and the support for dynamic allocation of resources.

Chapter 3 draws upon related work in to supporting software processes in a virtual environment, and discusses concepts and design choices related to visual programming which VRPML is based upon. Specifically, flow-based visual language issues are discussed in terms its suitability for supporting the modelling and enacting of software processes.

Chapter 4 discusses and justifies the detailed semantics of VRPML based on discussion from Chapter 3 and the requirements from Chapter 2. Additionally, issues related to the enactment model are also explained.

In Chapter 5, a more detailed account of the case study, the ISPW-6 problem, is given with a step-by-step solution expressed in VRPML. Moreover, additional experience with VRPML to support the modelling of the waterfall software development model is also summarised.

Chapter 6 discusses issues relating to implementation. How enactment can be achieved is also addressed. A prototype implementation is discussed examining the feasibility of VRPML and allowing further insights into VRPML.

In Chapter 7, the evaluation of VRPML is presented. In particular, the ISPW-6 and the waterfall development model solutions offered by VRPML in Chapter 5 are revisited and evaluated against those offered by other PMLs whenever possible. Additionally, the novel features offered by VRPML are also highlighted.

The conclusion of this work is given in Chapter 8, where the achievements, contributions and problems are summarised. Additionally, the main research hypothesis is revisted and the usefulness of VRPML is debated. Conclusions are drawn from the experience gained from this work and the significance of findings along with a consideration for future work.

Chapter 2 – Survey of Software Process Modelling Languages

The previous chapter established the need to model a software process using a process modelling language (PML). As an analogy, this need is much like the need to use a programming language to provide a model of a solution of a particular problem. However, there is a subtle difference between a PML and a programming language which lies in terms of their computational models.

Unlike the familiar computational model in computer science where sequences of operations specified by computer programs are automatically executed by the central processing unit or interpreting system, the computational model demanded by PMLs is somewhat different. This is because PMLs require some operations to be executed by (error-prone) software engineers using some defined software tools, while other operations are suitable for automatic execution (e.g. automation of routine operations). As such, developing an ideal PML to support such a computational model poses a challenge to researchers in computer science. In response to this challenge, many PMLs have been developed and described in the literature over the last fifteen years.

Despite many potential advances, the use of PMLs in industry has not been widespread [42]. As PMLs could form a vital feature for future software engineering environments, it is useful to reflect on the current achievements and shortcomings, and to identify potential areas of omission. It is also useful to explore issues emerging from related research areas, the adoption of which could improve the applicability and acceptance of PMLs. Given such potential benefits, this chapter presents a critical analysis of existing PMLs identifying each language's strong points and weaknesses, thereby forming guidelines for the future design of PMLs.

It is sometimes difficult to separate features of a PML from the Process Centred Software Engineering Environment (PSEE) supporting that PML. In this thesis, the focus is placed on features of a PML and its underlying semantics irrespective of whether some of those features may be supported by the PSEE.

This chapter begins by giving an overview of the concepts and terminology that will be used. Then, the following section discusses the salient features of PMLs and uses these features to form a taxonomy for PML. Using the taxonomy, the chapter then surveys and analyses existing PMLs. This analysis is then used to provide justification for the development of VRPML, the language that is the basis of this thesis. Finally, this chapter closes by providing a short summary.

2.1 Overview

A software process can be viewed as a partially ordered set of activities that must undertaken by software engineers to manage, develop, maintain and evolve software systems. To allow better control of a particular software process, a model of that process (called a process model) can be created using a PML making the process explicit and open to examination. A process model can show deficiencies and inconsistencies in the current software process that would otherwise be obscured, hence making it easier to analyse the process and suggest improvements.

The use of the term model to refer to a software process can sometimes be misleading. Here, the use of the term model is based on the analogy between PMLs and conventional programming languages in the sense that conventional programming languages can be thought of as capturing a model of a (software) system. Generally, a model is an abstraction of some particular subject (or phenomenon) using some form of medium (e.g. mathematics). In most cases, a model captures only important attributes which influence the subject being modelled (e.g. its behaviour). In the context of this research, the word model can have a subtle meaning when it is used to refer to a software process (i.e. process model). While a process model can also be seen as an abstraction of a particular software process, it can also be the actual software process, that is, when that process model is being enacted to support the actual software development.

Based on the work of Greenwood *et al* [34], two types of process model can be considered: passive and active. A passive model is no more than just a description of the actual software process. An active model, on one hand, is also a description of the actual software process. On the other hand, an active model is the actual

software process when it is enacted. Therefore, it is enactment, the key feature of a PML, which turns a passive model into an active one.

While active and passive models of a software process are closely related as they can be both specified by a PML, it is the active model that is of interest in this thesis. Only through active models can PMLs be used to support the activities of software engineers in a direct manner. In order to investigate active models of software processes, it follows that only enactable PMLs will be considered in this thesis. Non-enactable PMLs are not neglected, as an appendix which briefly describes some of their main features, is provided towards the end of this thesis.

Having drawn the boundaries of this research work, the next section develops an analogy of software processes with software engineering in an effort to define the terminology that will be used in the rest of this thesis.

2.2 Terminology

As discussed earlier, the terminology used to refer to the act of modelling and enacting software processes has a strong analogy with many concepts in software engineering. The use of PMLs to develop process models is analogous to the use of programming languages to develop software systems. Also, enactment of the process model is much like the execution of a software system. However, it must be stressed that despite many similarities between PMLs and conventional programming languages, the latter are inappropriate for modelling software processes as they lack features to accommodate the special requirements of a software process.

Similar to the development of software systems, the acts of modelling and enacting software processes are also performed in a defined lifecycle. For obvious reasons, this lifecycle is often called the *process lifecycle* (similar to the term software engineering lifecycle). The fact that the act of modelling and enacting of software processes involves a defined lifecycle raises an issue related to the scope of coverage of PMLs. In software engineering, when the terms implementation language, specification language or design language are used, they are directly related to specific parts of the software development lifecycle. However, in the context of software processes, the distinction between specification languages,

design languages and implementation languages is not made as they are all referred to as PMLs. As a consequence, some approaches described in the literature offer different PMLs for different phases of the process lifecycle. Nevertheless, as highlighted in the previous section, the main emphasis of this research is on the enactable PMLs which are used in the implementation phase of the process lifecycle.

In terms of the supporting environments, Software Development Environments, Software Engineering Environments and Integrated Development Environments are the environments where the software engineers can edit, compile, debug and run software. In the same manner, a *Process Centred Software Engineering Environment* (PSEE) is the environment where the "process engineers" can define, modify, analyse and enact a process model. In addition, a PSEE normally provides some form of task management facilities, for example, in terms of a to-do-list which captures the specific tasks assignments for software engineers. This is not a feature normally found in the software developments environments.

Continuing the analogy with conventional software development, a process model will need to change during the course of its lifecycle. These changes are referred to as *process evolution*. Typically, the enactment of a process model is long-lived, depending on the duration of the development project. As a result, it is likely that the process model itself requires modification while it is being enacted. The reasons for modifications might include the fact that the process model itself is faulty and ineffective, or the context in which the process is applied has changed, for example, due to shifting business targets or introduction of new technology, methods or tools.

Evolution of any process model is subjected to a *meta-process*, that is, a prescribed process for changing and maintaining a process model. Similar to other activities in software engineering, the meta-process involves joint intervention of humans and tools working on the software process and obeying a set of policies (i.e. as defined by the specific organisation). As will be seen later in the chapter, because it is crucial to address evolution, the meta-process will also need to be expressed using a PML.

Having defined the necessary terminology, the next section focuses on the main issues relating to PMLs. In particular, the salient features of PMLs are identified and discussed leading to a taxonomy for PMLs.

2.3 PML Issues

In an attempt to find an ideal PML for the modelling and enacting of software processes, there has been much research into different language paradigms. Many of the language paradigms have been adapted following experiences from existing approaches in software engineering applied in the context of a software process.

There have already been a number of attempts to classify these different language paradigms. For example, Liu and Conradi [52] identify five categories of PML language paradigms:

- Active Database PMLs – PMLs which relies on database triggers employing Event-Condition-Action rules as the basis for the language.
- Rule-based PMLs – PMLs which exploits rule-based planning techniques or blackboard architectures in the language.
- Graph/Net PMLs – PMLs which utilizes graphs or Petri Nets.
- Process Programming PMLs – PMLs which define a process model as a computer program based on a general purpose programming language.
- Hybrid PMLs – PMLs which fits into more than one of the above categories.

Although with different headings, Lonchamp [53] proposes a similar classification of PMLs, consisting of the following categories:

- Graphical PMLs – PMLs which provide a graphical syntax.
- Net-oriented PMLs – PMLs which utilises nets such as Petri nets.
- Procedural PMLs – PMLs which adopts a procedural programming language.
- Object-oriented PMLs – PMLs which utilises some features from object oriented languages (e.g. objects and inheritance).
- Rule-based PMLs – PMLs which exploits rules-based techniques mainly based on Prolog.

- Multi-paradigm PMLs – PMLs which utilizes more than one of the above categories.

The slight difference between the classification from Liu and Conradi with that from Lonchamp is that the latter include object-orientation. This object-oriented categorisation was appropriate at the time because many authors of PMLs were beginning to incorporate features from object-orientated languages as part of their PMLs.

Based on the classifications from Liu and Conradi and Lonchamp, Huff [38] identifies four different categories of PML language paradigms:

- Non-executable PMLs – PMLs which provide defined syntax but without executable semantics.
- State-based PMLs – PMLs which uses hierarchical state machines, Petri nets, or formal grammars as the basis of the language.
- Rule-based PMLs – PMLs which rely on a rule-based approach (i.e. based on Prolog) or database triggers (i.e. based on Event-Condition-Action (ECA) rules).
- Imperative PMLs – PMLs which rely on a model of computation whereby software processes are modelled as step by step sequences of commands.

The notable difference between the work of Liu and Conradi, Lonchamp and Huff is that Huff includes the non-executable category of PMLs. Essentially, the outcome of Huff's non-executable categorisation is that graphical high-level notations such as Integration Definition and Function Modelling (IDEF0) and Entry condition, Tasks, Verification and Exit Criteria (ETVX) can also be considered as PMLs because of their syntactic abilities to express a software process [38]. It should also be noted that Huff's non-executable category of PMLs does not implicitly imply that PMLs in other categories are enactable. This is because some PMLs can belong to more than one category.

A more recent classification of PMLs is that of Ambriola *et al* [5]. This classification breaks away from the earlier classifications, as PMLs are classified

according the process lifecycle that they support rather than being based on their language paradigm. The main categories are:

- Process Specification Languages (PSLs) – used in the specification phase of the software process, and typically make use of formal notations.
- Process Design Languages (PDLs) – used to support the design phase of the software process.
- Process Implementation Languages (PILs) – used to support the implementation phase of the software process.

In addition to providing a clear distinction between PMLs according to the part of the process lifecycle that they support, the other potential advantage of using the process lifecycle as the main classification of PMLs is that it is straightforward to distinguish between PMLs which are capable of supporting active and passive models regardless of the language paradigm.

As discussed earlier, regardless of which classification is used, some PMLs can fit into more than one category. This is because some PMLs combine different language paradigms, and therefore cannot be classified under one particular category. Additionally, such an overlap may also occur because the scope of coverage of a PML can be very large covering many aspects of the process lifecycle from the requirement phase to implementation.

As far as assessing the existing PMLs, much research has already appeared. Lonchamp [53] reports some results of evaluating PMLs using a set of questionnaires given to the authors of each PML. The questionnaires covered:

- The modelling approach – language constructs used to express activities and their pre-conditions and post-conditions as well as ordering constraints and parallelism, input and output artefacts, and roles.
- The underlying language paradigm.
- The PML tools support (e.g. editors, compilers).
- The enactment capability, the meta-process and the evolution support.

The resulting assessment, however, is limited to the PMLs developed under the European research consortium called Process Modelling Techniques Research (PROMOTER).

Building from the work of Lonchamp, Ambriola *et al* [5] define an assessment grid for evaluating both PSEEs and PMLs. As far as evaluating PMLs is concerned, the assessment grid covers:

- The PML scope of coverage – the part of the process lifecycle the PML supports.
- The underlying language paradigm.
- The modelling approach.
- The support for modularity, composition and reuse.
- The mechanism for process enactment and evolution.
- The PML tool support (e.g. including the support for synchronous and asynchronous communication tools).

Using the assessment grid, Ambriola *et al* evaluate some selected PMLs.

Finally, like Ambriola *et al*, Conradi and Jaccheri [21] also define an assessment grid in the forms of requirements for PMLs and PSEEs, identifying the primary and secondary process elements (i.e. in terms of what constitute a software process) that a PML and a PSEE need to support. In the context of this research work, only the primary process elements are considered because they constitute the requirements of PMLs whilst the secondary process elements will be ignored as they constitute the requirements of PSEEs. According to Conradi and Jaccheri, the primary process elements consist of:

- Activities, their pre-conditions and post-conditions, ordering constraints, and parallelism.
- Input artefacts and output artefacts as products.
- Human and their roles representation.
- Tool support.
- Evolution support.

Using these primary process elements as the basis for the requirements of PMLs, Conradi and Jaccheri offer an evaluation of existing PMLs giving particular emphasis to the PMLs in the PROMOTER context.

As has been shown, there have been a number of attempts to classify and characterise the requirements of PMLs. However, one aspect perceived to be lacking in the previous classifications of PMLs is consideration of the human dimension which relates to the issues surrounding the software engineers (or process engineers) who create the process models and initiate enactment as well as the software engineers who are subjected to process enactment.

Paradoxically, human dimension issues have always been a major concern in research into software processes [6, 19, 39, 61]. However, current trends in the way software engineers work (e.g. cross organisational boundary, geographically and temporally distributed locations) suggest that much more could be done to address these issues. These issues include: providing support for process engineers in terms of utilising visual syntax; enactment within a virtual environment; supporting user and process awareness; process visualisation; virtual meetings; as well as reflecting that support in the features provided in a PML [70].

Capitalising on these issues relating to the human dimension and building from the earlier characterisations and requirements of PMLs, Table 2-1 presents an alternative characterisation of PMLs which forms a taxonomy for PMLs [72]. This taxonomy for PMLs differs from the earlier work mainly by the inclusion of the human dimension issues.

	PML Characteristics
Modelling	Support for expressing both sequential and parallel activities as well as their constraints
	Support for expressing input and output artefacts
	Support for role representations
	Support for expressing and invoking external tools
	Support for abstraction and modularisation
Enactment	Support for enactment in a distributed environment
	Support for dynamic allocation of resources
Evolution	Support for reflection
Evaluation	Support for collection of enactment data
Human Dimension	Support for visual notations
	Support for user awareness
	Support for process awareness
	Support for process visualisation
	Support for virtual meetings

Table 2-1 Taxonomy for PMLs

As depicted in Table 2-1, issues that a PML needs to address include: modelling; enactment; evolution; evaluation; and human dimension. Each issue will now be discussed in detail.

Modelling

Derniame *et al* [26] have identified the following constituent parts of a process model which therefore have to be represented in a PML:

- Activities – Activities are any actions performed by software engineers or by computers to achieve certain set goals. Examples of an activity can include high level design or compilation of a program. In term of enactment, activities can be sequential or parallel and are always associated with artefacts (described below) and sets of pre-conditions and post-conditions. In addition, depending on its needs, an activity may be performed by a single software engineer (e.g. modify a design) or collaboratively performed by a group of software engineers (e.g. review a design).

- Roles – Roles identify the skills required for performing a particular activity. In many cases, software engineers may assume many different roles based on their skills.
- Artefacts – Artefacts represent the inputs to and the outputs from an activity. Generally, artefacts are referred to, produced or maintained when an activity is performed. Examples of artefacts include: design documents; source code; and object code. Because artefacts are potentially manipulated by many software engineers when performing their activities, artefacts often require some associated access rights. Access rights ensure that artefacts are manipulated in accordance with the pre-conditions or post-conditions of an activity. For example, the activity "modify design" may require that design documents are updated. But, to perform such an update to the design documents, software engineers may need to refer to the source code with the provision that they may do so without being able to change the code – that is, the code must have a read-only access right.
- Tools – Tools are external programs which are needed either to transform artefacts or to support inter-person communication (e.g. email, video conferencing program) which is seen as an important aspect of collaborative activities such as software processes [68].

The last three constituents of a process model from the bulleted list above are often referred to as *resources*. Resources are made up of software engineers assuming certain roles, artefacts, and tools which are assigned to activities before they can be undertaken. Clearly, resources are needed by an activity for it to be carried out.

In addition to accommodating activities and resources, a process model must be reusable, that is, a process model developed in one project can also be reused in other projects. In order to achieve reuse of process models, a PML needs to support abstraction and modularisation. Through modularisation and abstraction, large process models for a particular project, for example, can be broken into a number of smaller process models (or modules). In turn, these smaller process models can be reused to form other process models for different projects.

In summary, the categories for PML modelling issues are:

i. Support for expressing both sequential and parallel activities and their constraints.

ii. Support for expressing input and output artefacts.

iii. Support for role representations.

iv. Support for expressing and invoking external tools.

v. Support for abstraction and modularisation of process models.

Enactment

In order to directly support the activities of software engineers, a process model needs to be enacted. Through enactment, a process model can assist automation, guidance and enforcement of software engineering practices and policies. Enactment of process models requires a PML that has executable semantics. As software processes often involve software engineers who may or may not be collocated, a PML also ought to support enactment of process models in a distributed environment.

Enactment of a process model raises an issue relating to resource allocations. Because software processes are highly dynamic, rarely can resources for a process model be completely specified ahead of time. For instance, the number of people assigned (as a resource) for a particular activity, and hence how many instances of a particular activity are created, must not be fixed since it will depend on the dynamic needs of a project. Therefore, it is desirable for a PML to support the dynamic allocation of resources. Here, the dynamic allocation of resources means that resources are allocated at the last moment, just as the activity is about to be started. Allowing dynamic allocation of resources as a feature of PML gives the process engineers the flexibility to consider the current needs of a particular project before deciding on the necessary resource allocation for a particular activity. As a consequence, because resources are allocated dynamically, enactment of a process model can commence even when resources have not yet been completely specified.

In summary, the categories for PML enactment issues are:

i. Support for enactment in a distributed environment.

ii. Support for dynamic allocation of resources.

Evolution

To handle its evolution in a controlled and integrated manner, a process model also needs to capture issues concerning the meta-process [18, 56]. As discussed earlier, the meta-process is in charge of maintaining and evolving the process model according to specific and desirable rules and procedures. Therefore, the data manipulated by the meta-process are part of the process model itself.

Because enactment of a process model is typically long-lived and subjected to unpredictable changes (e.g. to cater for new needs arising from the current enactment), there is a particular need for the PML to provide a mechanism to allow the meta-process to be able to access the process model even though it is running. It has been suggested that reflection, a feature of a PML which allowed enactable code to be manipulated as data, provides such a suitable mechanism [5, 8, 18, 21, 56]. With reflection, the meta-process can be modelled and enacted as part of the process model itself. As a result, evolution of the process model (and evolution of the meta-process itself) can be achieved dynamically, and be supported by the meta-process. It follows that while an enactable PML without a reflective facility can be used to support modelling and enacting of software processes, it may not be able to support an integrated process model consisting of both the software process and the meta-process, and to support dynamic changes to both during enactment.

In summary, the sole category for PML evolution issues is:

i. Support for reflection.

Evaluation

If a PML, through enactment of the process model, is being used to guide or enforce software engineering practice, it is vital that issues of importance are measured in some way so that evaluation of the process can take place. This is especially important to provide support for software process improvement. Thus, a PML ought to provide relevant software metrics, although little work is reported in the literature [32].

In summary, the sole category for PML evaluation issues is:

i. Support for collection of "enactment" data.

Human Dimension

Because software processes are carried out primarily by people, it is necessary that human dimension issues are considered. The human dimension can cover issues for the process engineers (or project managers) who create the process models and initiate enactment (e.g. facilitating the construction and comprehension of process models) as well as issues for software engineers who are subjected to process enactment (e.g. supporting software engineers at work).

In terms of the human dimension issues surrounding the process engineers who create the process model and initiate enactment, it is obviously desirable to have a process model which is simple and easy to understand. This places a requirement on a PML to be intuitive in its syntax and semantics. It is generally believed that this can be obtained to a certain extent by adopting a visual syntax and notation. The reason is that "pictures" are normally thought to readily relate to the cognitive part of the human brain as compared to text. Employing visual notations in a PML helps create an easy to use yet expressive language, thus making a PML more acceptable and accessible. There are a number of visual PMLs discussed in the literature, as will be seen in the next section.

In terms of the issues surrounding the software engineers who are subjected to process enactment, it is desirable that enactment of a process model provides some form of awareness in terms of providing information about other parts of the model such as other users and other activities. In the literature, the importance of awareness has been established in the field of Computer Supported Cooperative Work (CSCW), a field of study which places emphasis on the nature of humans working together collaboratively to achieve a common goal as well as on the possibilities of technology to support and improve individual and group efficacy. Clearly, there is a need to include support for awareness as a feature of a PML since enactment of a process model also involves collaborative work similar to CSCW. As discussed earlier, the support for awareness seems increasingly relevant in line with the growing trends in the way software engineers work in

geographically and temporally distributed locations (e.g. software designers in London, reviewers in Washington and programmers in New Delhi) and across organisational boundaries.

There are two types of awareness a PML must support [72]. They are:

- User awareness – User awareness is providing knowledge about other group members involved in the cooperative system. In general, having user awareness can often encourage informal interaction. Such informal interaction is normally useful if people are working on shared artefacts (as in a typical software process).
- Process awareness – Process awareness is providing knowledge about the tasks in their working contexts, for example in terms of what the previous task was, what the next task is and what needs to be done to move along as well as what resources are required. Typically, having process awareness is valuable as it gives a sense of where and how the pieces fit together into the whole picture. Additionally, process awareness should also make people aware of tasks not only involving themselves but also others. Such awareness may help improve the process – for instance, people can plan and anticipate their work loads as needed to meet the project deadline. One way to enhance process awareness is to provide support for visualisation of the process model. This seems to be a useful feature to have in a PML as software processes can be very complex and full of subtleties. Process visualisation can provide multiple views of the same process with different perspectives which, in turn, enhances human intuition about the tasks they are involved in.

Apart from supporting awareness and visualisation, there is also a need for a process model to be able to accommodate meetings as they are an important characteristic of software engineering. In fact, in the context of supporting software development over distributed locations, it is desirable to for a process model to accommodate virtual meetings, that is, meetings that are held online. Accommodating virtual meetings could reduce costs if meetings would otherwise have to be held face to face. Thus, a PML needs to be able to specify and support virtual meeting as part of the process model.

In summary, the categories for PML human dimension issues are:

i. Support for visual notations.

ii. Support for user awareness.

iii. Support for process awareness.

iv. Support for process visualisation.

v. Support for virtual meetings.

2.4 Analysis of PMLs

This section provides a detailed analysis of existing PMLs using the characteristics of PMLs identified in the previous section. For each PML, this section presents: a brief description of the language; an analysis of PML issues related to the modelling, the enactment, the evolution, the evaluation, and the human dimension; and some overall observations about the language.

2.4.1 MSL

Marvel Strategy Language (MSL) [45, 46], the PML for the PSEE called Marvel, has been developed at the Columbia University. MSL uses a rule-based approach to support software processes. Artefacts involved in the software processes are stored in an object-oriented database.

In MSL, each activity in a software process is encapsulated in a rule that can be invoked from the software engineer's menu during enactment. An MSL rule has a name, typed formal parameters, and three optional constructs:

1. *Condition*: A condition specifies the bindings which are used to locate artefacts in the object-oriented database, and the pre-conditions which must be true before the activity can be initiated.

2. *Activity*: An activity may specify one of the following: a *tool envelope* for interfacing with an external tool to operate on the artefacts specified by bindings in the condition; an activity to be performed offline; or the selection of another rule set defining other activities [45].

3. *Effects*: Effects specify a list of post-conditions which are automatically selected and asserted based on the outcome of the activity.

Rules are stored in files called *strategies*. Usually, rules with similar effects are grouped together into a single strategy file.

When the user (e.g. software engineer) requests a particular activity, MSL rules defined in the process model are traversed by the runtime system using backward and forward chaining. Backward chaining occurs when the precondition of a selected activity is not met, and MSL looks for another activity it can perform to generate post-conditions that would satisfy the pre-conditions of the current activity. If successful, that activity is enacted. Otherwise, similar backward chaining is re-applied. The result of backward chaining is either satisfaction of the action or a notification to the user that the activity cannot be enacted. Forward chaining allows opportunistic execution of an activity automatically as soon as its pre-conditions are satisfied as a result of prior steps being performed.

Analysis of MSL

In terms of the modelling support, MSL does not provide support for all characteristics identified in Table 2-1 earlier. In particular, MSL does not support the representation of roles. Support for modelling of activities and their constraints are indirectly achieved through definitions of rules and specified in strategies. External tools can be specified in the tool envelope. Parallel activities are supported in MSL and are handled automatically by the Marvel PSEE. Modularisation and abstraction are supported using strategies.

In terms of enactment support, the process model expressed by MSL can be enacted in a distributed client-server environment. No support is provided for dynamic allocation of resources.

As far as evolution support, MSL does not provide support for reflection. Also, no support is provided in terms of evolution, evaluation, nor the human dimension issues identified earlier.

Observation on MSL

In general, a process model expressed in MSL is non-trivial and can be difficult to comprehend. This is because the dependencies between activities must be extracted from the rules and their constraints defined in the strategies. To get the overall view of the activities, each one of the strategies has to be analysed.

2.4.2 HFSP

Hierarchical and Functional Software Process (HFSP) [47], a PML developed at the Tokyo Institute of Technology, takes a functional approach to supporting software processes. In HFSP, software processes are described as a collection of activities which are characterised by their inputs and outputs. When the activities are complex, they can be hierarchically decomposed into sub-activities together with the definitions of their inputs and outputs, until an atomic level where such activities can be mapped onto an external tool invocation or (atomic) manual operations.

A HFSP process model consists of three main sections:

1. *Type declaration section*: A type declaration section allows definition of the attributes and primitive types used in the model. Apart from standard string and number types, HFSP also allows enumerations and user-defined types.

2. *Tools description section*: A tool description section permits definition of tools which will be used in the particular process model along with lists of their inputs and outputs. In HFSP, tools definition can be thought of as representing the most atomic activities in a software process apart from the atomic manual activities themselves.

3. *Activity section*: An activity section defines all of the activities involved in the particular process model. As discussed earlier, activities are defined along with their inputs and outputs, and can be decomposed. In HFSP, several alternative decompositions of a particular activity can be defined, each of which carries its own pre-conditions (called decomposable conditions).

In HFSP, the basic flow of controls of activities (i.e. their enactment and sequencing) is based on the availability of their inputs. If an activity can be decomposed and several alternative decompositions are defined (discussed above), only one of the possible decompositions will be selected based on the satisfied decomposable transition. However, in the case that the selected decomposition fails to produce the required outputs, HFSP allows backtracking which re-evaluates other decomposable conditions for alternative decompositions. Nonetheless, backtracking does not perform restoration of any artefacts which have been changed.

Additionally, HFSP also allows support for iterations, for example, to support repeated modification of a design until the design is approved. In such a case, the outputs of the current iteration become the inputs of the next iteration.

Analysis of HFSP

In terms of the modelling support, HFSP does not directly provide support for all characteristics identified in Table 2-1 earlier. In particular, HFSP does not support representation of roles. Artefacts are represented as part of the inputs and outputs of an activity; they may also be defined using user-defined types. Invoking of external tools is supported in HSFP. Activities are characterised by their inputs and outputs and may be decomposable. Activities in HFSP are intrinsically concurrent; unless there are explicit dependencies on their inputs, they may be evaluated in parallel.

In terms of enactment support, HFSP supports enactment in a distributed environment. However, no support is directly provided for dynamic allocation of resources although process models (and resource allocation) can be evolved while enactment is taking place through reflection.

For evaluation support, HFSP does not provide any support for the collection of enactment data. Finally, in terms of human dimension support, HFSP also does not provide any support for the characteristics defined earlier.

Observation on HFSP

The support for modelling of software processes provided by HFSP is very high-level in the sense that only what goes into and out of an activity is specified and not how an activity is performed. This is due to the fact that decomposition of an activity can be mapped onto either an external tool invocation or onto (atomic) manual operations. In the case where decomposition is mapped onto an external tool invocation, the software engineer can be given some flexibility with respect to how the activity can be performed – for example, they can use their autonomous judgement depending on the needs of the activity. While in some creative activities, such as design, flexibility may be sometimes beneficial, a more prescribed approach might be desirable, for example to guide junior software engineers.

On a positive note, although HFSP is a textual language, it is still relatively easy to comprehend an HFSP process model. This is because HFSP supports activity decomposition which can be easily traceable by following the inputs and outputs dependencies of each activity.

2.4.3 FUNSOFT Nets

FUNSOFT nets [29], the PML for the PSEE called Melmac, is based on high-level Petri nets called Predicate/Transition nets. To understand the basic features of FUNSOFT nets, it is necessary to introduce some elementary Petri Nets concepts.

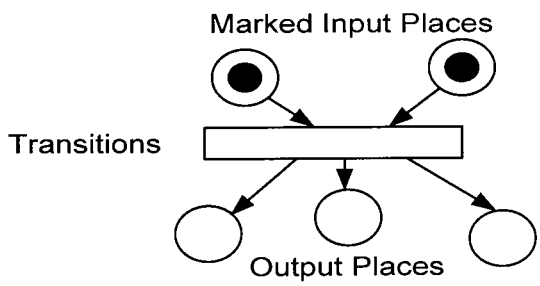


Figure 2-1 A General Petri Net

Figure 2-1 depicts a general Petri net. A Petri net is a directed graph with two types of nodes: *places* and *transitions*, using circles to denote places and rectangles to denote transitions. Places and transitions are connected together by arcs. Arcs from

places to transitions designate input places whilst arcs from transitions to places designate output places. The firing rule of a Petri net depends on the distribution of tokens in the input places, called *marking*. A transition in a Petri net will be fired when all of its input places are marked. When a transition is fired, tokens are then distributed to the output places of that transition.

In FUNSOFT nets, activities are represented as transitions called *agencies*. Agencies can be decomposed to contain other FUNSOFT nets. Artefacts are represented as tokens and are stored in places called *channels*. Within channels, tokens are typed. Some predefined types include: boolean; integer; real; string; text and binary. Other types such as structures are also supported using a C-like syntax.

A number of input and output firing rules, which control the way transitions fire, can be associated with agencies. The actions or constraints associated with each agency can also be specified using C-like statements. An agency can be associated with a modification point before enactment commences to allow dynamic refinement of an activity. Upon reaching a modification point, enactment may be suspended to allow the behaviour of that activity to be refined. If the refinement involves only changes to the topology of the net, FUNSOFT nets allow the changes to occur on-the-fly, that is, while enactment is taking place. If the refinement involves redefining types in channels or actions in agencies, enactment must be suspended as recompilation is necessary. Once the refining process is completed, enactment is allowed to commence from that modification point.

Analysis of FUNSOFT Nets

In terms of the modelling support, FUNSOFT nets do not seem to provide support for most of the characteristics identified in Table 2-1 earlier. The representation of roles is supported as a user-defined role type in a channel connected to an agency representing an activity. The agency can only be executed if such a role is available and willing to conduct the activity. Support for modelling of activities and their constraints are achieved through agencies, and artefacts and tools are specified in the channels and expressed by a Petri net. Lastly, modularisation and abstraction facilities are also supported by FUNSOFT nets through decomposable agencies.

In terms of enactment support, process models expressed by FUNSOFT nets can be enacted in a distributed environment. Although supporting (limited) on-the-fly evolution in terms of changing the topology of the net, FUNSOFT nets do not provide support for reflection.

No support is provided for the collection of enactment data. In terms of human dimension support, FUNSOFT nets use both visual notations (based on Petri nets) and textual syntax based on the C-like syntax.

Observation on FUNSOFT Nets

Although based on visual syntax, it is still quite difficult to comprehend the process models expressed by FUNSOFT nets. One reason is that FUNSOFT nets employ an extension of Petri nets with a number of different firing rules (attached to agencies). Understanding these different firing rules requires evaluation of transitions, tokens and places. As a PML, this is somewhat of a low-level way of supporting modelling and enacting of software processes, although FUNSOFT nets do support abstraction of process models through agencies.

2.4.4 SLANG

SLANG [9], a PML for the PSEE called SPADE, is also based on Petri nets. However, the semantics of SLANG are defined by high-level Petri nets called Entity Relations (ER) nets. The main addition that ER nets add to conventional Petri nets is the ability to incorporate timing as the criteria to fire transitions.

In addition to the usual Petri nets syntax, SLANG provides an interface defined in terms of input and output *places* (as *entry* and *exit* points), input and output transitions (as *entry* and *exit* actions) as well as *shared places* to allow sharing of data, all of which are connected by a sets of input and output arcs. In fact, an interface serves as a SLANG modularisation facility; an interface may be decomposed to show the overall internal SLANG net structure.

Besides the normal places and the shared places, SLANG also defines *user places*. Normal places and shared places represent place holders for tokens consisting of typed artefacts stored in an object oriented database, while user places represent

place holders for tokens consisting of internal messages generated as a consequence of external events occurring within the PSEE.

In SLANG, transitions designate events. Transitions firing depends upon the availability of tokens and the defined guards (explained below). Additionally, timing information, such as the time interval within which an event may or must occur, can also be specified by associating time-stamps with tokens and time changes with actions (described below). In addition, SLANG also allows transitions to be textually augmented with scripts consisting of three parts:

1. A header containing an event's name and typed parameters which must match the types of the transition's input and output places.

2. A guard containing a boolean expression which checks the firing rules. A guard may be thought as the pre-condition for enabling a transition.

3. A set of actions that performs some computation on the input tokens to produces some output tokens.

There are two types of transitions in SLANG: white transitions and black transitions. White transitions are similar to procedures in a general purpose programming language – they receive some input parameters (though the defined guards that need to be satisfied) and predefined statements are executed in sequence to produce some output parameters. Black transitions, unlike white transitions, allow invocation of external tools. The combinations of black transitions and user places allow SLANG to have some control over the events generated by external tools. This capability allows SLANG to support automation at external tool levels, for example, by detecting events such as the opening or the closing of external tools.

Analysis of SLANG

In terms of the modelling support, SLANG seems to provide support for most of the characteristics identified in Table 2-1 with the exception of the representation of roles. In SLANG, activities and their constraints are modelled as sets of actions associated with transitions. Tokens represent typed input/output artefacts which are stored in an object-oriented database. Being a Petri net based PML, SLANG

naturally supports parallelism. Finally, modularisation and abstraction facilities are also supported in SLANG through interfaces.

In terms of enactment support, SLANG supports enactment in a distributed environment. However, SLANG does not directly support dynamic allocation of resource but process models (and resource allocation) can be evolved while enactment is taking place through SLANG's reflection facility.

No support is provided for the collection of enactment data. As far as the human dimension support, SLANG only supports visual notations based on Petri nets (and augmented with textual scripts).

Observation on SLANG

Although based on visual syntax, SLANG is a complex language with numerous extensions to the original Petri nets. Like FUNSOFT nets, this makes a process model expressed in SLANG quite difficult to understand. Despite being a highly expressive PML, the abstraction level for supporting modelling and enacting of software processes is still somewhat low. This is because a user may still need to work at the level of the Petri net (i.e. in terms of transitions, tokens and places) to understand the process model expressed by SLANG.

On a positive note, SLANG excels in its support for process evolution and the meta-processes. This is achieved through SLANG's reflective feature which allows enactable process models to be accessed as code and data.

2.4.5 LIMBO and PATE

LIMBO and PATE, the two PMLs for the PSEE called OIKOS [4, 55], have been developed at the University di Pisa, Italy. Because both LIMBO and PATE exploit the idea of coordination to support a software process as inspired by Linda [33], it is worth describing the basic idea from Linda.

In Linda, coordination is based on *tuples* and *tuple spaces*. A tuple consists of a set of variables or values. A tuple space can be viewed as a distributed shared memory into which tuples can be inserted or removed. Each tuple in a tuple space is produced by some executing thread and it remains in the tuple space until some

other thread consumes it. When a thread requests specific tuples which do not exist, the thread may be suspended until those tuples are made available.

Borrowing from Linda, LIMBO and PATE support a software process by exploiting a reactive system based on threads (called *agents*) communicating using shared tuple spaces called blackboards. It must be stressed that the word agents in this case carries the meaning of execution threads.

LIMBO and PATE originated from Extended Shared Prolog, and adopt a rule-based approach to the support modelling and enacting of software processes. Using Ambriola's classification discussed in section 2.3, LIMBO is the specification language whilst PATE is the implementation language. LIMBO and PATE are closely related in that it is possible to obtain a PATE process model by successive refinement of a LIMBO specification. In the context of this thesis, because LIMBO serves as a specification language for PATE, LIMBO will not be discussed further.

In PATE, a process model is constructed in terms of a hierarchy of agents. Each agent is connected to its own blackboard when the agent is activated. Agents react to the presence of tuples (mainly Prolog facts) on their blackboards by removing tuples, and inserting tuples into their own blackboards or other blackboards that they know of through a customisable service provided by the PSEE.

The behaviour of an agent is defined by a *theory* consisting of a set of action and reaction patterns along with a sequential Prolog program (called the *Knowledge base).* Each pattern defines a stimulus and response pair. The stimulus consists of a *Read guard* and an *In guard.* The response consists of a *Body* and a *Success set.* Optionally, a *Failure set* can also be specified.

A pattern can fire when the tuples (in terms of Prolog facts) in the agent's blackboard satisfy the read and the in guards; these tuples will be consumed and removed from the agent's blackboard. Whenever several patterns can fire, one is chosen non-deterministically and the specified actions are performed (i.e. the related pattern body and the sequential Prolog program will be executed). The execution of the pattern body and the sequential Prolog program is achieved in such a way that no side effects are allowed to the agent's blackboard whose pattern is fired (as this can only be done by the success set or the failure set).

The most common use of the sequential Prolog program in terms of supporting a software process is to invoke external tools to manipulate artefacts. Finally, depending on the outcome of the execution of both the pattern body and the sequential Prolog program, some tuples (i.e. a success or a failure set) will be inserted back onto the blackboard whose name is specified by that pattern. This sequence of pattern firing can then go on for other agents until the enactment is completed.

Blackboards can be dynamically created or destroyed by agents during the course of enactment. Creation of a new blackboard can be achieved by an agent inserting an *activation goal* (i.e. Prolog facts) along with a *termination condition* (also Prolog facts) and a list of connected agents in its own blackboard. Destruction of a blackboard occurs when the success set or the failure set matches with the blackboard termination condition also expressed as Prolog facts, as a result of firing a pattern. In this case, all the tuples in the blackboard will disappear and all connection to agents will be aborted.

Analysis of LIMBO and PATE

In terms of modelling support, the modelling of activities and their constraints are indirectly supported by specifying the theory of each agent (i.e. the action and reaction patterns and the knowledge base). This can be achieved either from the LIMBO specification (and later refined to PATE) or directly in PATE. How role representation is supported in PATE is not clear. Tools can be invoked in pattern bodies or in the knowledge base. Artefacts are accessed in terms of identifiers through some standard services provided by the PSEE. Parallel activities can be readily supported because agents are effectively executing threads communicating using multiple tuple spaces. However, modularisation and abstraction of process models are not supported in PATE.

In terms of enactment support, a process model expressed in PATE can be enacted in a distributed environment. However, PATE does not support dynamic allocation of resources.

In terms of evolution support, PATE does not support reflection. No support is provided for the collection of enactment data nor for the human dimension issues.

Observation on LIMBO and PATE

The abstraction level for supporting a software process provided in LIMBO and PATE is low. The reason is that the project engineers who model a software process need to work at the agent level, essentially specifying threads for enabling enactment, in addition to specifying the process model. Often, this can be difficult because of the need to consider for deadlock situations.

Nonetheless, the approach taken by LIMBO and PATE is not without merit. In particular, agents specified by PATE are independent and highly decoupled. Thus, it may be possible to support (limited) process evolution through a meta process while a process model is being enacted, such as changing the tools assignments in the knowledge base, without affecting other agents (including those agents whose patterns are firing).

2.4.6 BM and PWI PML

Base Model (BM) and Process Wise Integrator Process Management Language (PWI PML) [65], the two PMLs for the PSEE called PADM [13], have been developed by the Informatics Process Group at the University of Manchester. BM adopts temporal logics semantics; PWI PML adopts object-oriented technology. Using Ambriola's classification, BM can be seen as a process specification language whilst PWI PML can be seen as a process implementation language.

Actually, BM and PWI PML are two compatible PMLs in the sense that it is possible to gradually refine the process model specified by BM into PWI PML. In doing so, a special tool called the BM stepper can be used to assist checking of the BM specification against the problem description and the refinement of the process model in PWI PML. Because this thesis concentrates on enactable PMLs, discussion of BM will not be developed further as it mainly serves as a specification language for PWI PML.

PWI PML is an object-oriented PML. The primary construct for supporting modelling and enacting of software processes is the *role*. The concept of a role in PWI PML carries a more subtle meaning than that defined in section 2.3. PWI PML defines two types of roles: *User Roles* and *System Roles*. A user role corresponds to

the identification of skills for performing a particular activity in a software process whilst a system role may correspond to some abstractions of an activity or a user role.

A software process model in PWI PML is a set of executing role instances connected by interactions (described below). A role is a subclass of the pre-defined PML Role class. Within the subclass the following properties must be specified for modelling and enacting software processes:

1. *Resources* describe the data objects (e.g. artefacts and tool definitions) belonging to the role. These data objects can be derived from some pre-defined classes in PWI PML.

2. *Assocs* provides references to the communication channels linking one role object to another.

3. *Actions* define the list of interactions and activities which are performed by the role. Actions may be thought of as sub-activities of a role. Each of these actions has a name, and is guarded by *when* conditions which must be satisfied before the action can be performed.

4. *Categories* contain conditions to determine the start and stop conditions of a role.

Enactment is achieved by instantiating roles, and a role may also instantiate other roles. Each role instance has a separate thread of control with its own local data. Role instances communicate using message passing via typed one-way asynchronous communication channels.

PWI PML also offers a set of interfaces known as *UserActions*. Normally, UserActions are specified as the first command in the role action properties discussed earlier. With the help of the PSEE runtime engine, UserActions allow the construction of *agendas*, essentially a to-do-list for the software engineers listing all the possible actions whose guard conditions are satisfied in the particular role that they play. If the software engineer selects among those possible actions in the agenda, the selected action is performed.

Analysis of BM and PWI PML

In terms of modelling support, the modelling of activities and their constraints are supported using the specification language BM and later refined into PWI PML. In particular, activities can be abstracted as a sub-class of the Role class, with each sub-activity representing some actions properties in the role. Parallelism between activities is supported as each role has its own separate thread. The representation roles described in section 2.3 are represented as user roles and are considered as resources of the system roles. Similarly, artefacts and tools are also considered as resources of the system roles. Finally, modularisation and abstraction in PWI PML are supported by the role definitions.

In terms of enactment support, PWI PML supports enactment in a distributed environment. In fact, the latest version of PWI PML also supports enactment over the World Wide Web [35]. However, PWI PML does not seem to directly support dynamic allocation of resources although user roles can be bound to system roles dynamically.

In terms of evolution support, PWI PML provides support for reflection. No support is provided in PWI PML for the collection of enactment data nor for the human dimension issues identified earlier. It is worth noting that although PWI PML is a text-based PML, it utilises a special graphical notation based on the Role Activity Diagram, that is, a diagram which shows interactions amongst roles along with the "exchange" of artefacts.

Observation on BM and PWI PML

The fact that system roles need to communicate through defined asynchronous channels raises an important issue relating to synchronisation and concurrency. It seems common in PWI PML to handle issues of synchronisation and concurrency directly. This can be seen, for example, by the need to specify the required communication channels inside the process model. This seems somewhat low-level for a PML to handle.

2.4.7 MERLIN

MERLIN [44] is a Prolog-like PML which has been developed by the University of Dortmund and STZ – Gesellschaft fur Software-Technologie mbH. In MERLIN, the act of modelling a software process is assisted by entity relationship diagrams (a type of diagram which mainly depicts dependencies amongst artefacts) and state charts (a special type of state transition diagram which depicts the allowable transitions of an activity or an artefact from its creation to its completion along with the conditions under which a transition may occur). Based on the created entity relationship diagrams and state charts, a process model is then mapped to Prolog rules and facts as a *knowledge base* about that particular process. A special kind of fact is used to describe roles (*work_on* and *responsibilities* facts), artefacts and tools (*document* facts) as well as activities (*task rules*). Similar to MSL discussed in section 2.4.1, enactment of a process model expressed by MERLIN is achieved by the PSEE runtime engine using a forward chaining mechanism to automate an activity that does not involve human intervention, and a backward chaining mechanism to select a particular activity for software engineers based on the role they play.

With the information gathered by the backward chaining mechanism, the MERLIN PSEE runtime engine incrementally builds a *work context* for the software engineers who perform the activity, in the form of a simplified entity relationship diagram which shows only the artefacts and tools necessary to complete the activity. In MERLIN PSEE, a software engineer may interact with this diagram in a hypertext manner to perform their work.

Analysis of MERLIN

In terms of modelling support, the modelling of activities and their constraints, roles, artefacts and tools are supported by specialised PROLOG facts. Parallel activities can be readily supported by the PSEE runtime engine. However, it is not clear how the modularisation and abstraction of process models are supported in MERLIN.

In terms of enactment support, the process model expressed by MERLIN can be enacted in a distributed environment. However, MERLIN does not support dynamic allocation of resources.

In terms of evolution support, MERLIN does not support reflection. For evaluation support, no support is provided for the collection of enactment data. Concerning human dimension support, although MERLIN is a textual language, state charts and entity relationship diagrams can be used to help construct the MERLIN process model. MERLIN provides limited support for awareness. Process awareness is supported but not user awareness. The support for process awareness is achieved by the MERLIN PSEE runtime engine which automatically builds a work context from the specified Prolog facts (utilising the backward chaining mechanism) in terms of only giving a software engineer the necessary artefacts and tools needed to complete the activity. Finally, MERLIN provides no support for process visualisation.

Observation on MERLIN

Although MERLIN employs state charts and entity relationship diagrams to assists the construction of the process model which is mapped to Prolog facts, it is still not clear how this mapping is done. A state chart and an entity relationship diagram may provide only a high-level view of the software process but because they do not have well-defined executable semantics to allow a direct mapping to Prolog, it may well be that the actual construction of the process model is achieved at the Prolog level.

Like other rule-based PMLs, the process model expressed in MERLIN is also non-trivial and can be difficult for humans to comprehend. This is because the dependencies between activities are not directly available and must be extracted from the rules (as defined by Prolog facts).

2.4.8 SPELL

SPELL, the PML for the PSEE called EPOS [20], is an object-oriented PML derived from the Prolog programming language. In SPELL, the main support for software processes is provided by two pre-defined classes: *TaskEntity* and

DataEntity. TaskEntity forms the root of the task type hierarchy whilst DataEntity forms the root of the data (or artefact) type hierarchy.

Every SPELL task type must be a subclass of TaskEntity which defines a number of predefined attributes that can be tailored to the needs of the process model. Among the important type level attributes within a task type are:

1. *Pre- and post-conditions*: The pre- and post-conditions attributes in a task type are divided into static and dynamic pre- and post-conditions. They are specified in first order predicate logic (as in Prolog). Static pre-conditions and post-conditions are constraints mainly used to build the network of tasks. This is accomplished by the runtime planner using forward and backward chaining. Dynamic pre-conditions and post-conditions are constraints asserted before and after task executions, and are used to dynamically trigger tasks.

2. *Code*: The code attribute defines the steps that are performed when the task is executed. A task's code (specified in Prolog) is responsible for satisfying the dynamic post-conditions. When the code attribute is empty (i.e. not specified), the task type is assumed to be composite – that is, the task is not performed by a specific piece of code, but rather by executing subtasks (explained below).

3. *Decomposition*: The decomposition attribute relates to the code attribute described earlier as it allows subtasks to be specified. In SPELL, the subtasks may also be a network of tasks. Like the parent task, the network of subtasks is also created by the runtime planner using the static pre- and post conditions discussed earlier.

4. *Formal*: The formal attribute permits the specification of the input and output artefacts required for the task.

5. *Executor*: The executor attribute allows the tools used in the task to be specified.

6. *Role*: The role attribute represents the role for the task.

In addition to these attributes, the TaskEntity class also defines a number of meta-level attributes and methods, essentially allowing further customisation of the TaskEntity class in terms of its execution. These meta-level attributes and methods will not be discussed here as they are mainly there to provide support for the reflection facility in SPELL. Nevertheless, one important aspect that can be specified at this level is triggers, which are special operations invoked before or after the occurrence of a method. Triggers specify the constraints defining when the trigger "codes" should be executed with respect to the method call. With triggers, various internal states of the task executions can be captured and modified if needed.

SPELL also defines a family of types derived from the DataEntity class for specifying artefacts. Typical types used for software processes are a text type and a binary type which can be further specialised to other types (such as c-source and object-file).

For process enactment in SPELL, the runtime support system provided by the PSEE consists of two parts: the runtime planner and the runtime execution manager. As discussed earlier, the runtime planner generates a network of tasks from the static pre- and post-conditions by utilising forward and backward chaining similar to MSL; the top-level network of tasks must be generated by the runtime planner before enactment commences but the detailed level network of subtasks can be generated incrementally. The runtime manager executes the given task network (by executing the specified code attribute in the task type) and works closely with the runtime planner such that, when a composite task is encountered, the runtime manager invokes the runtime planner to detail out that composite task based on its static pre- and post-conditions. In this way, SPELL allows the detailed network of tasks to be generated only when it is needed.

Analysis of SPELL

In SPELL, support for modelling of activities is provided by inheriting the TaskEntity class and specifying the various pre-defined attributes discussed earlier. Parallel activities are supported in SPELL and are handled automatically by the runtime planner and the runtime manager. The activity constraints are defined by

the static and dynamic pre- and post-conditions. Role and tool abstractions are supported through the role and executor attributes of the TaskEntity class. Artefacts are typed and stored in an object-oriented database called EPOS-DB. Modularisation and abstraction facilities are also supported through the code and decomposition attributes of the TaskEntity class.

In terms of enactment support, the process model expressed by SPELL can be enacted in a distributed environment. SPELL does not directly support dynamic allocation of resources but process models (and resource allocation) can be evolved while enactment is taking place through SPELL's reflection facility. No support is provided for the collection of enactment data nor for the human dimension issues identified earlier.

Observation on SPELL

Process models expressed in SPELL are quite difficult to comprehend. This is because the overall task structures are not directly available. To have a clear picture of the overall task structures, it is necessary to analyse the constraints defined by both the static and dynamic pre- and post-conditions.

On a positive note, although lacking in human dimension support, SPELL seems to cover the other PML characteristics very well. In particular, SPELL excels in its support for process evolution and the meta-processes. This is because SPELL allows manipulation of methods and attributes at the meta-type level, hence, enabling it to capture and modify various internal states of the task executions.

2.4.9 MASP/DL

Model for Assisted Software Process Description Language (MASP/DL) [14], the PML for the PSEE called ALF, has been developed at Nancy University, France. In MASP/DL, a software process model is modelled as a number of "fragments" called the *MASP descriptions*. Each MASP description models a software process as five components:

1. *The Object Model* describes the data model representing artefacts.

2. *The Operator Model* represents an abstraction of the actual activities which a software engineer needs to perform in a software process, in terms of operator types. Operator types allow an individual activity to be described in terms of pre-conditions and post-conditions along with a definition of tools.

3. *The Characteristics* specify a set of consistency constraints on the process state which are maintained by the runtime engine during the course of enactment; if they are violated, an exception condition is raised.

4. *The Rule Model* defines some trigger reactions for predefined events that occur during process enactment. The events include database operations (e.g. read, write) and other user defined events.

5. *The Ordering Model* specifies the flow of control of the operators. Operators may be executed in parallel, alternatively or sequentially.

Because a MASP description is generic, it must be instantiated (and compiled to a special format called IMASP) before enactment can be achieved. Instantiation of MASP description corresponds to the assignment of actual tools and artefacts in the operator and object models. It should be noted that a particular software process model can consists of a number of MASP descriptions (or fragments) with each description defines a number of related activities. So, before enactment can be achieved, each MASP description must be instantiated (and compiled) individually. In this manner, instantiation and enactment may interleave so that the part of the software process that has already been enacted may be taken into account to instantiate a further part.

Analysis of MASP/DL

In terms of modelling support, the modelling of activities and their constraints in MASP/DL are supported in the operator and rule models. Parallelism of activities is supported in the ordering model. The representation of roles is not supported at the PML level as this is achieved at the PSEE level. Artefacts are described in the object model. Tools are described in the rule model along with the defined triggers. Finally, modularisation and abstraction are supported by the MASP descriptions.

For enactment support, MASP/DL supports enactment in a distributed environment. However, MASP/DL does not really support dynamic allocation of resources. This is because resources for each activity defined in a particular MASP/DL description must be allocated before enactment of that MASP/DL description can commence, although such allocations can be made based on the outcome of the instantiation and enactment of the earlier parts.

For evolution support, MASP/DL does not support reflection. Also, no support is provided for the collection of enactment data nor for the human dimension issues identified earlier.

Observation on MASP/DL

The approach of treating a particular process model as number of independent fragments in MASP/DL can be both a burden and a feature.

- As a burden, it may be inconvenient and difficult to support a large scale software process. This is because in a large scale software process, one may need to have literally hundreds of MASP descriptions. Each of these individual MASP descriptions then will have to be individually instantiated and compiled to IMASP before enactment can be achieved.
- As a feature, by manipulating the abstraction and modularisation of fragments, MASP/DL can allow enactment even when the overall instantiation has not yet been completed (although at least one of the MASP descriptions of a particular process model must already be instantiated).

2.4.10 ADELE and TEMPO

ADELE and TEMPO [12] are two PMLs for the PSEE called ADELE-TEMPO, which is a commercial database product. To understand how the modelling and enacting of software processes is supported by the first PML, ADELE, it is necessary to understand the basic architecture of its PSEE. The PSEE consists of a centralised database which provides long transaction support for artefacts involved in the software development project. Each participating software engineer in the project is provided with their own Work Environment by the PSEE, where artefacts (organised in terms of files and directories) can be checked out. A software process

is then modelled in ADELE as a set of defined events and triggers on the artefacts in the Work Environment.

ADELE events and triggers are based on the event-action-condition (ECA) rules. The ECA rules have the following form:

IF ON event EVALUATE condition EVALUATED {true | false} THEN action

Triggers may fire on any of four events – *PRE*, *POST*, *ERROR* and *AFTER* – with respect to a particular activity. In each of the events, some actions (e.g. in terms of some database transactions or invoking of external tools) can be specified. PRE triggers are evaluated at the beginning of an execution of an activity. If successful, the specified action is executed. Similarly, POST triggers are evaluated after the completion of an activity. ERROR triggers are considered when a transaction fails and AFTER triggers are applied after a transaction succeeds.

According to Belkhatir [12], because ADELE is too low level and difficult to understand, the second PML called TEMPO was developed. TEMPO defines a process model based on the concept of role and connection (described below). Although a user role is supported [54], the concept of role in TEMPO carries a subtle meaning than the one defined in section 2.3. A role allows redefinition of behavioural properties of an artefact based on the defined work context, that is, the operations that can be done on that artefact and the rules that control these operations. Roles, in turn, are connected to other roles through a defined connection, essentially the synchronisation protocol based on temporal-event-condition-action (TECA) rules (i.e. extended ECA rules with timing dependencies).

Analysis of ADELE and TEMPO

In terms of modelling support, the modelling of activities and their constraints is indirectly supported in TEMPO by defining roles and their connections. In ADELE, modelling of activities and their constraints are also indirectly supported although at a very low level in terms of the ECA rules. Tools can be invoked from the action part of the TECA rules in TEMPO (and ECA rule in ADELE) as a result of a condition being fulfilled. In both ADELE and TEMPO, artefacts are defined as objects in the ADELE database. In TEMPO, parallel activities are supported by

connections defined by TECA rules (and ECA rules in ADELE). Finally, modularisation and abstraction are only supported by TEMPO (through the abstraction of roles).

In terms of enactment support, process models expressed by TEMPO and ADELE can be readily enacted in a distributed environment. However, neither TEMPO nor ADELE supports dynamic allocation of resources.

In terms of evolution support, TEMPO and ADELE do not support reflection. For evaluation support, no support is provided for the collection of enactment data. As far as the support for the human dimension, TEMPO provides (limited) support for process awareness. In TEMPO, process awareness is achieved by giving the work context of a task. No support is provided for user awareness, process visualisation, and virtual meetings identified earlier.

Observation on ADELE and TEMPO

The approach taken by ADELE and TEMPO to support modelling and enacting software processes based on database triggers is somewhat low-level. This is because the complete picture of the overall software processes is not directly available and must be inferred through the defined ECA rules.

While the database triggers in TEMPO and ADELE may be low-level, it may be straightforward to implement support for a transactional history of artefacts and their traceability. In turn, such capabilities could be manipulated to provide support for the collection of enactment data through measurement and evaluation of the actual process enactment.

2.4.11 APPL/A

APPL/A [62], the PML for the PSEE called Arcadia, has been developed at the University of Colorado. Being based on ADA, APPL/A inherits many features from that language including its type system, module definition style (package), and task communication paradigm (rendezvous). To support a software process, APPL/A extends the ADA programming language with *shared persistence*

relations, *concurrent triggers on relation operations*, *enforceable predicates on relations*, and *transaction-like statements*.

Relations are syntactically similar to ADA package definitions and package bodies. Within a relation, persistent storage of data may be defined. Triggers are similar to ADA tasks and hence are capable of handling multiple threads of control. Unlike ADA tasks, triggers automatically react to events related to operations on the data defined in a relation. Enforced predicates are boolean expressions which act as post conditions on the operation of a relation; no operations may violate the enforced predicate. Transactions-like statements control access to relations and may affect the enforcement of predicates.

Analysis of APPL/A

In terms of the modelling support, APPL/A does not provide support for all of the PML characteristics identified in Table 2.1. Modelling of activities and their constraints are supported in APPL/A via relations with enforceable predicates. Parallel activities are supported by exploiting triggers. No direct support is provided for role representations, artefacts, and tools; rather these can be represented using the ADA type mechanism. Abstraction and modularisation in APPL/A is mainly based on the ADA procedures and packages.

In terms of enactment support, it is not clear whether or not APPL/A supports enactment in a distributed environment. Additionally, APPL/A does not support dynamic allocation of resources.

In terms of evolution support, APPL/A does not support reflection, only offline process evolution as recompilation is necessary. Finally, no support is provided for the collection of enactment data nor for the human dimension issues identified earlier.

Observation on APPL/A

The abstraction level provided by APPL/A for supporting modelling and enacting software processes is somewhat low. This is because, in addition to the specific constructs to support modelling and enacting software processes, APPL/A inherits

many general purpose constructs based on ADA which appear to be unnecessary. For example, the ADA tasking mechanism and rendezvous seems to be superseded by triggers. However, it must also be stressed that the presence of general purpose constructs based on ADA is not without merit. Constructs such as ADA types help APPL/A provides support for the representation of roles, tools and artefacts.

2.4.12 Dynamic Task Nets

Dynamic Task Nets [37], the visual PML for the PSEE called Dynamite, has been developed at Aachen University of Technology in Germany. Dynamic Task Nets described a software process as graphs consisting of nodes representing tasks connected together with arcs (called *relations*). There are three types of relations:

1. *Control-flow relations* impose an acyclic ordering of the activities to be enacted.

2. *Data flow relations* are used for data (mainly artefacts) transmitted between connected tasks.

3. *Feedback flow relations* are used to enable feedback from a successor task back to its predecessor.

There is also another relation supported by Dynamic Task Nets called *successor relations*. Unlike the three relations described above, successor relations refer to nodes rather than arcs. When a task is augmented with a successor relation, that task is said to have multiple versions. What this means is that when such a task has to be reactivated (e.g. as a result of feedback relations), a new task version may be created depending on whether the previous version of the task has completed or not. If the task has already completed, a new version of the task is created requiring a new assignment of a software engineer. If the task has not yet been completed, the runtime system automatically updated the tasks with the new version of the artefacts.

In Dynamic Task Nets, tasks and their corresponding relations can be defined dynamically. The behaviour of each individual task (called a *task net)* can be customised in the sense that a task can execute even when its predecessor tasks have not completed (called *simultaneous concurrent engineering*) or when certain

input artefacts are available. However, the task completion can only be allowed if predecessor tasks have already been completed. In Dynamic Task Nets, this customisation is achieved by modifying the enactment conditions defined in the PROGRESS specification, which itself is an executable graph rewriting system that Dynamic Task Nets map to for achieving enactment.

Analysis of Dynamic Task Nets

In terms of modelling support, the modelling of activities and their constraints are supported by the task nets and their relations, and can be customised at the PROGRESS specification level. The representation of roles is not directly supported at the PML level. Artefacts are considered as part of the input and output interface to a task. Tools abstraction is not supported directly although can be defined at the PROGRESS specification level. Finally, modularisation and abstraction are supported by the nodes themselves in the sense that each task net can be decomposed into other smaller task nets (although Dynamic Task Nets does not visually differentiate between decomposable task nets and atomic ones).

In terms of enactment support, Dynamic Task Nets support enactment in a distributed environment. Because the creation of task nets in Dynamic Task Nets is dynamic, it naturally supports dynamic allocation of resources. In terms of evolution support, Dynamic Task Nets does not support reflection.

No support is provided for the collection of enactment data. In terms of human dimension support, Dynamic Task Nets employs a visual PML. No other support is provided to address the other human dimension issues identified earlier.

Observation on Dynamic Task Nets

Although based on a visual notation, the process model expressed by Dynamic Task Nets can still be difficult to comprehend. In addition to employing three different kinds of arcs (consisting of control-flow relations, data flow relations and feedback flow relations), Dynamic Task Nets also do not distinguish between decomposable nodes and atomic ones. This can make it difficult to trace out the execution of the process model.

The main advantage of Dynamic Task Nets lies in the support for on-the-fly process evolution – that is, process models can be evolved while enactment is taking place. With on-the-fly process evolution, Dynamic Task Nets are able to support the dynamic behaviour required in a PML such as dynamic creation of tasks and dynamic allocation of resources.

2.4.13 LATIN

Language to tolerate Inconsistencies (LATIN) [22], developed at Politecnico di Milano, Italy, is a PML for the PSEE called SENTINEL. LATIN describes a software process as a global part and a set of task types. A global part contains global variables, a global invariant (described below), the declaration of the task types that will be instantiated during enactment, and the description of the *main task*. When the process enactment starts, an interpreter for the main task is created. All other tasks are instantiated by the main task or, in turn, by previously instantiated tasks (i.e. their predecessors).

A task type is composed of the following parts:

1. *A header* defines the name of the task type along with its parameters lists. Instantiation of a task type constitute assigning the initial state of the task instance.

2. *An import section* defines all variables imported from other task types.

3. *A declaration section* declares the local types and variables. The basic data types in LATIN can be integer, real, string, boolean, enumerated, as well as user defined data types such as records and sets.

4. *An export section* lists the names of all the variables exported to other tasks.

5. *An init section* lists all initial values assigned in terms of resource assignments during instantiation.

6. *A set of transitions* which govern the actual enactment of task type instances (described below). Transitions can be associated with invoking of external tools.

7. *A set of invariants* which serves as special conditions that must hold true in any state of the activity described by the task type.

The behaviour of a task type is described by transitions. Transitions are further characterised by a precondition, called an *ENTRY*, and a body. The ENTRY defines the conditions which fire the corresponding transition whilst a body defines *actions* (e.g. invoking a tool) and *value assignments* (e.g. updating variables) through an *EXIT* clause.

LATIN actually offers two types of transitions: *normal transitions* and *exported transitions*. A normal transition is automatically executed by the runtime system as soon as an ENTRY evaluates true. An exported transition is executed upon the request from the user even if its ENTRY evaluates to false. In such a case, a transition is said to *fire illegally*. The outcome of such a firing is that enactment can now be allowed to deviate from the specified process model, hence introducing *inconsistencies*. In LATIN, enactment can continue as long as the invariants of that task type still hold. But if one of the invariants is violated, enactment is suspended and a *reconciliation activity* is started to allow reconciliation of the actual process and the process model. In such a situation, the PSEE runtime system automatically performs *pollution analysis* which gives some analysis of the deviations from the defined process model and the identification of some polluted data (mainly artefacts) caused by such deviations.

Analysis of LATIN

In terms of modelling support, the modelling of activities is supported as a set of task types. The constraints for each activity are specified by the transitions. Task types may execute in parallel. The representation of roles is not directly supported at the PML level. Artefacts can be supported by the user defined types. Tools can be specified and invoked in the transition's action. Finally, modularisation and abstraction are supported by each individual task type.

In terms of enactment support, LATIN supports enactment in a distributed environment. Although task types can be dynamically instantiated during enactment by the main task or the previously instantiated task, LATIN does not support dynamic allocation of resources. This is because resources need to be

assigned to the task types as part of their init section before the overall enactment commences as LATIN does not provide any feature which allows assignment of resources while enactment is taking place.

As far as evolution support, LATIN does not support reflection. For evaluation support, no support is provided in LATIN for the collection of enactment data. Finally, no support is provided for the human dimension issues identified earlier.

Observation on LATIN

The main feature of LATIN is that it supports deviation from the process model by allowing a start or a completion of any activities to occur even when their pre-conditions or post-conditions have not been met. This feature can be exploited to give software engineers the flexibility of exercising their own judgement, based on the (dynamic) needs of the current development project, as to whether or not to deviate from the process models. While giving such flexibility may be an important human dimension issue to be considered by a PML, a closer study reveals that allowing too many deviations can be problematic as it may be difficult to guide and prescribe tasks especially involving junior software engineers. Additionally, allowing too many deviations also puts extra overhead on the overall tasks as extra steps are needed for reconciliation.

2.4.14 JIL

JIL [63], developed at the University of Massachusetts at Amherst, is a PML derived from experiences in developing APPL/A. The main construct in JIL is the *step*. A JIL step represents a step in a software process, that is, a task which a software engineer or a tool is expected to perform. A JIL process model can be viewed as a composition of JIL steps. The elements that constitute a JIL step include:

1. *Object and declarations section* consists of the declaration of artefacts used in the step (consisting of ADA-like types).

2. *Resource Requirements section* specify the resources needed by the step, including people, software and hardware.

3. *Sub-steps set section* provides a list of sub-steps that contribute to the realisation of the step.

4. *Proactive control specification section* defines the order in which sub-steps may be enacted. This is achieved through special JIL keywords such as ORDERED, UNORDERED and PARALLEL.

5. *Reactive control specification* of the conditions or events in response to which sub-steps are to be executed. This is specified through special JIL keyword such as REACT.

6. *Pre-conditions, constraints and post-conditions section*: A set of artefact consistency conditions that must be satisfied prior to, during, and subsequent to the execution of the step.

7. *Exception handler section*: A set of exception handlers for local exceptions, including handlers for artefact consistency violations (e.g. pre-condition violations).

Apart from local exception handlers which allow a process to react within its own scope, JIL also supports global exception handlers. Global exception handlers allow a process to react to an exception in another process by treating such exceptions as events which can be handled directly by the reactive control specification.

Analysis of JIL

In terms of modelling support, the modelling of activities is supported by a composition of JIL steps. The constraints for each step can be specified as pre- and post-conditions. The step ordering constraints (e.g. parallel activities) can be supported using specialised constructs (e.g. *ORDERED, UNORDERED, PARALLEL*). The representation of roles is not directly supported at the PML level. Artefacts can be supported by the ADA-like types; JIL also allows artefact consistency to be checked as a pre- or post-conditions of a step. Tools can be invoked directly through the reactive control specification. Finally, modularisation and abstraction are supported by the JIL step.

In terms of enactment support, it is not clear whether or not JIL supports enactment in a distributed environment. Also, JIL does not support dynamic resource allocation. As far as evolution support is concerned, no support is provided for reflection.

No support is provided in JIL for the collection of enactment data nor for the human dimension issues identified earlier.

Observation on JIL

Despite being a complex PML, JIL has a number of strong points. In particular, JIL excels in combining proactive and reactive controls as well as allowing local and global exception handling. Proactive control allows one or more process steps to be imperatively programmed in a JIL step. Reactive control permits responses to stimuli or events by executing one or more process steps. Global exception handling allows one process to react to an exception raised in another process whilst local exception handlers allow a process to react within its own scope.

2.4.15 Little JIL

Little JIL [66], also developed at the University of Massachusetts at Amherst, is a PML based on JIL. Little JIL is a visual PML, and maintains the notion of *a step* from JIL. A process model in Little JIL can be viewed as a tree of steps whose leaves represent the smallest specified unit of work and whose structure represents the way in which this work will be coordinated. As an illustration, Figure 2-2 displays the Little JIL step notation.

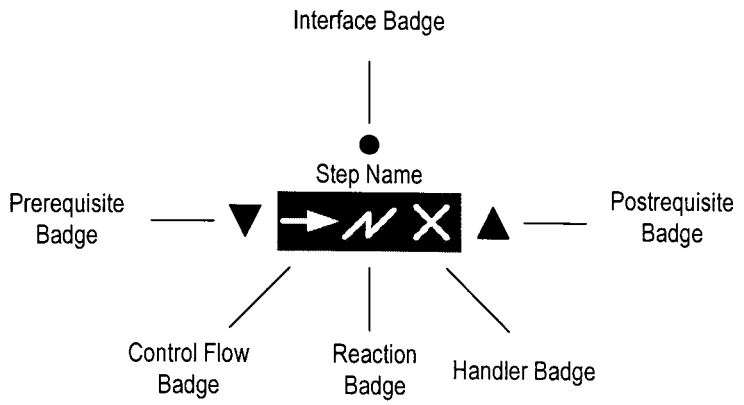


Figure 2-2 Little JIL Step Notation

As shown in Figure 2-2, the Little JIL step notation has a number of components which include:

1. *Step name section*: Every step must be given a name.

2. *Interface badge section*: Resources needed in the step are carried through this badge. In Little JIL, resources include the assignment of software engineers, permissions to use the tools, as well as artefacts. Resources are also typed and always associated with some access rights.

3. *Pre-requisite and post-requisite badge section*: Here, pre-conditions and post-conditions for the step are specified.

4. *Control-flow badge section*: Little JIL allows four types of control-flow to be specified in a step. They are *sequential*, *parallel*, *choice* and *try*. Sequential and parallel control-flows allow sequential and parallel steps to be specified. Choice control-flow allows some choices of alternative steps to be specified in a step. Try control-flow is associated with handler badges (described below) to allow an exception to be caught.

5. *Handler badge section*: Handler badges are used to indicate and fix exceptional errors during enactment. In Little JIL, exceptions are passed up the process model tree until a matching handler is found.

6. *Reaction badge section*: Reaction badges are a form of reactive control similar to JIL. A reaction badge is always associated with a message which

is generated in response to some events. Because a message is global in scope, any execution step can receive the message and act accordingly if matches are found.

In terms of its enactment, a step goes through several states. Normally, a step is *posted* when assigned to a software engineer, then it progresses to a *started* state, and eventually it will be in a *completed* state. If a step fails to be started as a result of resource exceptions being thrown, a step may be *retracted* (and potentially reposted) or *terminated* with exception.

Analysis of Little JIL

In terms of modelling support, the modelling of activities is supported by the Little JIL steps. The constraints for each step can be specified in the pre-requisite and post-requisite badges. Parallel activities can be defined by selecting the proper control-flow badge. The representation of roles is not directly supported at the PML level. In Little JIL, artefacts are typed and have associated access rights. Like artefacts, tools abstraction is also associated with an interface badge. Finally, Little JIL does not support modularisation and abstraction of the process models.

In terms of enactment support, Little JIL supports enactment in a distributed environment. However, Little JIL does not seem to support dynamic resource allocation as identified in section 2.3, although the authors claim such support is provided (discussed below). In terms of evolution support, Little JIL does not support reflection.

No support is provided in Little JIL for the collection of enactment data. In terms of human dimension support, Little JIL employs visual notation. No support is provided for process awareness, user awareness and process visualisation.

Observation on Little JIL

In the current implementation of Little JIL, dynamic allocation of resources is achieved by the runtime system dynamically (and automatically) searching for software engineers available for assignment to a step [67]. Nonetheless, dynamic resource allocation is much more than just allocating software engineers. The

allocations of other resources (e.g. artefacts and tools) must also be dynamic. This is because these resources may also change due to hardware or software upgrades. Because assignments of such resources are fixed in Little JIL, this makes it hard to support such this dynamic behaviour without evolving the process model.

On a positive note, Little JIL step notation is very intuitive. This makes it straightforward to understand the process model expressed by Little JIL.

2.4.16 CSPL

CSPL [15], developed at the National Chiao Tung University in Taiwan, is a PML that adopts an ADA95-like syntax. Being based on ADA95, CSPL inherits many features from that language including its type system, module definition style (package), and task communication mechanism. Additionally, CSPL adds a number of predefined types and extensions to enable the modelling of software processes which include:

1. *Event* type and *inform* statements: The event type allows description of an event status of an activity (e.g. approved, completed). The value of an event derived from event type can be asynchronously assigned by CSPL inform statements.

2. *Doc* type: Doc Type, the base type of all object types in CSPL, allows the description of artefacts and their associated attributes, which can be extended by inheritance.

3. Work assignment statements are CSPL statements which allow activities, tools and roles to be assigned to one or more software engineers.

4. Communication related statements allow synchronization and ordering of tasks with other tasks, similar to the ADA95 rendezvous.

5. Program Units allow assignment of a human to a role (through a *Role Unit*), assignment of an actual tool to a tool (through a *Tool Unit*), and description of dependencies amongst artefacts (through a *Relation Unit*).

To support enactment, the CSPL compiler translates the process model expressed in CSPL into a UNIX shell script.

Analysis of CSPL

In terms of modelling support, the modelling of activities and their constraints are supported by the ADA95-like task specification. Parallel activities are supported through the communication related statements essentially ordering the tasks specified in CSPL. The representation of roles is supported by the role unit. Artefacts can be supported by the ADA-like types. Tools can be defined through the tool unit. Finally, modularisation and abstraction are supported by utilising package specification.

In terms of enactment support, CSPL supports enactment in a distributed environment. However, CSPL does not support dynamic resource allocation.

In terms of evolution support, CSPL does not provide support for reflection – only offline process evolution is supported [16]. For evaluation support, no support is provided for the collection of enactment data. In terms of the human dimension, no support is provided for the issues identified earlier.

Observation on CSPL

As it has a blend of both a general purpose programming language and some specific syntax for the modelling and enacting of software processes, CSPL can be a powerful and expressive PML. Furthermore, the fact that CSPL supports a form of object-orientation can be also useful for the usual reasons (e.g. inheritance which can specialise types).

2.4.17 EVPL

Extended Visual Planning Language (EVPL) [36], developed jointly at the University of Waikato and University of Auckland in New Zealand, is a PML derived from the Visual Planning Language (VPL) [64] available commercially from FUJITSU ICL.

Naturally, many features of EVPL are based on VPL. This includes the notion of *stages* which form the main construct of EVPL. In EVPL, a process model (called *Work Plan*) consists of stages connected together by arcs carrying events. Parallel

activities are supported using *AND* and *OR* stages. The AND stage waits for all events from connected stages to arrive before generating a single event, while the OR stage allows an event to leave as soon as one of the incoming events arrive. Each stage in EVPL has an identifier and is always associated with a role (called the *stage role*). This identifier allows stages to be identified and reused by other process models. EVPL also includes visual notations to describe how the task is to be performed in the form of tools and artefacts used in each stage and the communication needed between stage roles (called the *Work Context*). In EVPL, the work context can be thought of as a visual task description as it does not have any executable semantics.

Enactment is achieved in EVPL through its event process mechanism. This mechanism is based on a sub-language called Visual Event Processing Language (VEPL). VEPL notations are based on *filters* and *actions*. Filters received events from stages, artefacts, tools or roles, or other filters and actions, try to match events against pre-conditions, passing them onto connected filters and actions if the match succeeds. When actions receive an event, they carry out one or more operations in response to the event. These operations may update information or generate new events, which may be detected and acted upon by other filters and actions.

Analysis of EVPL

In terms of modelling support, the modelling of activities is supported by the EVPL stages. The constraints for each stage can be specified in VEPL as filters and actions. Parallel activities can be supported by the AND and OR stages. The representation of roles is supported by associating a stage with a role. Artefacts and tools execution is supported in the action filters. Finally, modularisation and abstraction of the process models is supported in EVPL as stages can be further decomposed to include other stages.

In terms of enactment support, EVPL supports enactment in a distributed environment. However, EVPL does not support dynamic resource allocation. In terms of evolution support, EVPL does not provide support for reflection – only offline process evolution is supported.

No support is provided in EVPL for the collection of enactment data, although EVPL's PSEE does allow tracking of the enactment history. Lastly, apart from adopting a visual notation, EVPL does not provide support for the other human dimension issues identified earlier.

Observation on EVPL

The unique feature of EVPL is that it adopts a separate visual language (VEPL) to describe the events related to a software process. Although VEPL may be useful as it allows wide variety of events to be supported (e.g. by manipulating filters and actions), it may put an extra burden on the developer of the process model who has to learn two separate languages.

2.4.18 APEL

APEL [24], developed at Laboratoire Logiciels Systemes, Reseaux in France, is a visual PML. The central construct in APEL is an *activity* of which there are two types: an *activity* representing a task for an individual; and a *multi-instance activity* representing a task for a group of people.

Visually, an activity provides an interface to define input and output artefacts as well as the roles involved a particular activity. Artefacts, activities and roles are typed and they are defined in a separate view using state Object Management Techniques (OMT) diagrams, essentially class diagrams with some defined relationships (e.g. is-a or has-a). The various states which artefacts and activities go through during enactment can also be represented using state transition diagrams.

In APEL, a process model is composed of a set of activities connected together by *control-flow* and *data flow* arcs as well as *And* and *Or* connectors which carry the usual semantics. Activities can be decomposed until atomic activities are reached. To achieve process enactment, APEL relies on the concepts of *event* and *event capture* which can be defined on activities or artefacts. An event and event capture are defined by pairs comprising an event definition and a logical expression. An event is captured by an activity or an artefact when it matches the event definition and the logical expression is true. All events in APEL are broadcast and they are generated automatically.

The notable feature of APEL is that activities and their sub-activities, as well as the flow of artefacts, are shown to the user during enactment (through *a desktop paradigm*) in order to give the sense of awareness (discussed below) about other activities. In addition, the user may also interact with the desktop paradigm to perform the activity. Finally, unlike other PMLs, APEL also supports measurement of the process model by employing the Goal Question Metric Model [10] essentially consisting of self-defined goals, questions related to the process models achieving that goals, and metrics to quantify such questions.

Analysis of APEL

In terms of modelling support, the modelling of activities and their constraints is supported in APEL by the activities and their event and event captures. Parallel activities can be defined by And and Or connectors. Roles, artefacts and tools are typed and also expressed as part of the process models using OMT diagrams. Finally, modularisation and abstraction are supported as activities can be further decomposed into sub-activities.

In terms of enactment support, APEL supports enactment in a distributed environment. No support is provided for dynamic resource allocation.

In terms of evolution support, APEL does not support reflection. Process evolution can only be achieved offline; this is assisted by a process state server which maintains the state of a process model's enactment.

In terms of evaluation support, APEL supports the collection of enactment data through adaptation of the Goal Question Metric Model. In terms of the human dimension, APEL provides (limited) support for awareness. Process awareness is supported but not user awareness. Process awareness is achieved by giving the work context of the overall activity and its sub-activities along with the flow of artefacts. Likewise, process visualisation can also be achieved in the same way. However, no support is provided for virtual meetings.

Observation on APEL

It can be difficult to comprehend the process models expressed in APEL even though it uses visual notations. One reason is that APEL uses many notational supports, for example, using OMT for describing the relationship between artefacts, role and tools as well as adopting state transition diagrams to depict the states of activities and artefacts. Furthermore, APEL also employs two types of arcs (the control-flow and data flow) whose semantics can be difficult to understand (i.e. based on events and event capture).

The main feature of APEL is that it is the first PML to support some form of measurement and evaluation of the process models and their enactment. Additionally, the other main feature of APEL is its support for process awareness and process visualisation through the desktop paradigm. However, a user needs to have some understanding of APEL to benefit from the desktop paradigm as it is given in the form of the process model expressed in APEL.

2.4.19 PROMENADE

Process-oriented Modelling and Enactment of Software Developments (PROMENADE) [57], developed at Universitat Politecnia de Catalunya Spain, is a PML derived from the Unified Modelling Language [59], a language for supporting the object-oriented analysis and design of software systems. In PROMENADE, software processes are modelled using predefined classes (called the PROMENADE *reference model*) consisting of:

1. *Document Class* represents artefacts involved in the software development.
2. *Communication Class* represents any document used for communication.
3. *Task Class* represents an activity in a software process.
4. *Agent Class* represents an entity playing an active part in a software process.
5. *Tool Class* represents any entity implemented through a software tool.

6. *Resource Class* represents any supplementary help provided during enactment of a software process (e.g. an online tutorial). It should be noted that the word resource in this case carries a different meaning to that used in section 2.3.

7. *Role Class* represents identification of the skills of software engineers.

In a process model, the connections between instances of these classes are expressed using UML association relationships. In PROMENADE, the most important class is the task class. The task class may be customised to include the definition of shell scripts, the definition of task parameters consisting of input and output artefacts and the definition of task pre- and post-conditions. A task class may also be further broken down to sub-classes consisting of other task classes, either by aggregation or composition.

PROMENADE has not yet provided support for enactment as work is still on-going to provide additional language extensions. Nevertheless, it is considered in this survey because it presents a unique approach of utilising a PML based on UML to support the modelling and enacting of software processes.

Process enactment is planned in PROMENADE by utilising its support for *precedence relationships*, which are textually expressed in each task class in the process model using a variation of the UML Object Constraint Language (OCL). Using the precedence relationships, the ordering of tasks can be defined. Additionally, the precedence relationships also allow enactment of some actions according to a plan, defined in terms of pre-conditions and post-conditions, and connections to other task classes. *Strong* precedence dictates that the current task must successfully complete before following task can be started. *Weak* precedence allows following tasks to start even if the current task has not yet been completed. PROMENADE is also being extended to provide support for reactive control-flows that is, enactment of some actions in response to events [57].

Analysis of PROMENADE

In terms of modelling support, the modelling of activities is supported by the class diagrams and their association relationship. The constraints for each activity are

specified in the class diagram as pre- and post-conditions expressed by using a variation of the UML Object Constraint Language. Parallelism can be expressed in PROMENADE by specifying the precedence relationship in each task class. The representation of roles and artefacts are supported by the role and artefact classes respectively. The abstraction of tools is supported by a tool class and an agent class. Tools also can be invoked directly through the shell script defined in the task class. Finally, modularisation and abstraction are supported by aggregations and composition.

In terms of enactment support, it is not clear whether PROMENADE will support enactment in a distributed environment, and dynamic allocation of resources. Concerning evolution, the authors claim that PROMENADE will adopt reflection.

No support is provided in PROMENADE for the collection of enactment data. In terms of human dimension support, apart from being a visual PML, no other support is provided.

Observation on PROMENADE

As PROMENADE is still in its early stage, it is not clear how process enactment can be supported. This is because UML, the language that PROMENADE is based on, does not have any executable semantics. Although it is claimed that enactment can be achieved by through augmenting PROMENADE with OCL-like scripts, this has yet to be demonstrated.

2.5 Discussion

Based on the analysis of PMLs given in the previous section, this section summarises the description of existing PMLs. Although only enactable PMLs are considered, some of the categorisations of PMLs (e.g. modelling support, evaluation support) should also be equally applicable to non-enactable PMLs, although this is not investigated further here.

Table 2-2 presents the digest of the analysis of PMLs discussed earlier. The table highlights those features that are common to PMLs, and those that are not, and

therefore identifies areas for research into PMLs that address new combinations of features which can be explored further.

LEGENDS Implemented feature: √ Non supported feature: x Not enough information:		Enactable Process Modeling Languages																		
PML Characteristics		MSL	HFSP	FUNSOFT	SLANG	PATE	PWIPML	MERLIN	SPELL	MASP	TEMPO	APPLA	DYNAMITE	LATIN	JIL	LITTLE JIL	CSPL	EVPL	APEL	PROMENADE
Modelling Support	Sequential and parallel activities as well as their constraints	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Input and output artefacts	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Role representations	x	x	√	x	x	√	√	√	x	√	x	x	x	√	√	√	√	√	√
	External tools	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Abstraction and modularisation of process models	√	√	√	√	x	√		√	√	√	√	√	√	√	x	√	√	√	√
Enactment Support	Enactment in a distributed environment	√	√	√	√	√	√	√	√	√	√		√	√		√	√	√	√	
	Dynamic allocation of resources	x		x		x		x		x	x	x	√	x	x	x	x	x	x	x
Evolution Support	Reflective feature	x	√	x	√	x	√	x	√	x	x	x	x	x	x	x	x	x	x	x
Evaluation Support	Collection of enactment data	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	√	x
Human Dimension Support	Visual notations	x	x	√	√	x	x	x	x	x	x	x	√	x	x	√	x	√	√	√
	User awareness	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Process awareness	x	x	x	x	x	x	√	x	x	√	x	x	x	x	x	x	x	√	x
	Process visualisation	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	√	x
	Virtual meetings	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Table 2-2 Analysis of PMLs

Referring to Table 2-2, it can be seen that existing PMLs are deficient in a number of the identified areas: human dimension issues; dynamic allocation of resources; collection of enactment data; and support for reflection. These deficiencies will be discussed next.

The first perceived deficiency, and perhaps the most important, in the surveyed PMLs is in terms of providing support for human dimension issues. As identified in Table 2-2, issues in terms of the support for visual notations, user awareness, process awareness, process visualisation, and virtual meetings seems to be neglected by most of the surveyed PMLs. Modelling and enacting of software processes requires much human intervention during its lifecycle – for example, process engineers (e.g. project managers) model the activities within a software process for enactment, and software engineers perform the activities during enactment. Therefore, it seems appropriate to place emphasis on the importance of human dimension issues.

The second perceived deficiency in the surveyed PMLs is in term of the support for dynamic allocation of resources. The motivation behind supporting dynamic allocation as a feature of a PML and its underlying semantics is to ensure that resources are allocated based on the dynamic needs of a particular project, and as late as when the activity is about to be started. In this way, the process engineer can be given flexibility to allocate resources based on the current situation.

The third perceived deficiency in the surveyed PMLs is in term of the support for the collection of enactment data. Only APEL provides this support, that is, based on the Goal Question Metrics [10]. By supporting collection of enactment data, systematic and objective evaluation of a particular process model can be made. In turn, this evaluation could be used as "indicators" to support process improvement.

Finally, the fourth perceived deficiency in the existing PMLs is in terms of providing support for reflection. The main reason that reflection is required as a feature of PML is to support evolution of a process model through a meta-process [5, 8, 18, 21, 56]. In doing so, enactment of the existing activities must not be affected. With reflection, the enacting process model can be accessed as data to be modified by the meta-process, hence allowing the evolution of a process model to occur while enactment is taking place.

The above analysis has identified a novel combination of features which existing PMLs do not provide, thus identifying an area for new PML research. This combination of features includes the support for:

- Visual notations
- User awareness
- Process awareness
- Process visualisation
- Virtual meetings
- Dynamic allocation of resources
- Collection of enactment data
- Reflection

It is the vision of this research to develop a new PML which addresses all of the aforementioned features. However, within a scope of a PhD, it would be difficult to investigate all of these features. Therefore, this research will only concentrate on the first six items from the list because they are perceived as the most novel and important aspects for supporting modelling and enacting software processes [69, 70], in line with the main research hypothesis discussed in Chapter 1. The last two items from the list will be considered as the scope for future work, discussed in Chapter 8.

To address the support for visual notations, user awareness, process awareness, process visualisation, and virtual meetings, one novel approach might be to develop a visual PML which exploits a virtual environment. An example use of a virtual environment in a similar context, in the sense that it involves collaborative work among many participants, can be seen in the field of Collaborative Virtual Environment (CVE) – a subset of CSCW. The attractive characteristics of a virtual environment, such as the support for three dimensional graphics, immersion and user interactions, provide a good reason to explore integration of a PML with a virtual environment. In fact, it is generally believed that a virtual environment may provide superior user interfaces to the traditional two-dimensional Windows Icon Menu Pointer (WIMP) user interfaces [23].

To address the support for the dynamic allocation of resources as a feature of PML, the current solution, based on the analysis of PMLs discussed earlier, is by utilising process evolution while enactment is taking place. This approach can be seen in the PMLs such as SLANG and SPELL which supports process evolution by utilising their reflection facilities. Similar support for dynamic allocation of resources using process evolution can also be seen in Dynamic Task Nets which utilises its executable graph rewriting system. While evolution is a necessary feature of PML, utilising it to support dynamic allocation of resources can be difficult and expensive to achieve because of the need to enact the corresponding meta-process to guide the evolution process. Therefore, it seems rather unjustified to exploit the process evolution scheme just to achieve dynamic allocation of resources.

In order to overcome the difficulty faced by the above approach, a solution might be to investigate a novel enactment model for a PML based on a resource exception handling mechanism, essentially defining how the run-time system behaves when failing to allocate resources in order to enact a particular activity in the process model. The key feature of the enactment model is that it needs to allow the process engineer to intervene and rectify the resource exception, hence achieve dynamic allocation of resources.

In short, the two possible solutions to the deficiencies identified earlier suggest a new visual PML with the following new features: it considers a virtual environment as a fundamental constituent that is part of the construction of the process model (e.g. features in the language) as well as being part of the runtime environment; and supports dynamic allocation of resources through its enactment model. It is the development of such a visual PML, called Virtual Reality Process Modelling Language (VRPML), which is the main topic of this research.

2.6 Summary

In this chapter, the desirable characteristics of PMLs have been identified, and a survey of enactable PMLs has been presented based on these characteristics. PMLs have also been analysed to provide insights into their strengths and limitations, thus providing requirements and justification for the development of VRPML.

Building on the material developed in this chapter, the next chapter looks into related work of exploiting a virtual environment for the modelling and enacting of software processes, as well as discussing related design choices for VRPML.

Chapter 3 – VRPML Design Issues

In the analysis and survey of PMLs in the previous chapter, some limitations were observed in existing PMLs in terms of their support for user awareness, process awareness and process visualisation as well as for dynamic allocation of resources. To address some of these limitations, a new visual PML called VRPML is to be investigated.

This chapter will examine a number of research issues related to the design of VRPML. The first issue that will be discussed is whether or not it is useful to exploit a virtual environment as a way of supporting a software process, in particular by looking at the related work described in the literature. Being a visual language, the second issue that will be explored in this chapter relates to visual programming in terms of what is the suitable computational model for VRPML. Finally, the third issue that will be examined is a novel enactment model to support the dynamic allocation of resources.

3.1 Software Processes and a Virtual Environment

Because enactment of a process model requires much human involvement particularly to undertake the assigned activities, there is obviously a need to address the human dimension issues. As discussed in Chapter 2, among the human dimension issues are the need to support process awareness, user awareness, and process visualisation. Prior work has looked at exploiting virtual environments as a way of addressing some of these issues.

Doppke *et al* [27] concentrate on support for visualisation of a software process within a PSEE. According to Doppke, there are four different types of mappings for representing a software process:

- *Task-Centred Mapping* – Using task-centred mapping, each activity in a software process corresponds to a room in a virtual environment. In this case, tools and artefacts are objects inside the room.

- *Agent-Centred Mapping* – Using agent-centred mapping, a software engineer's workspace is mapped to a room. In this case, the workspace acts as a synthesizer of the multiple activities to be carried out by the software engineer. Similar to task-centred mapping, tools and artefacts are objects inside the room.
- *Resource-Centred Mapping* – Using resource-centred mapping, a room corresponds to resource instances within the software process. Being in the resource's room indicates possession of an instance of that resource. Similar to both task-centred mapping and agent-centred mapping, tools and artefacts are objects inside the room.
- *Artefact-Centred Mapping / Product-Centred Mapping* – Using artefact or product-centred mapping, artefacts are mapped onto rooms. The main difference between this mapping and other mappings is that dependency relationships between artefacts can be expressed as part of the arrangement of the rooms, for example, if sub-products of the artefacts are defined then they are represented as separate rooms connected to the parent product room either by exits or by containment. Similar to other mappings, tools are objects inside the room.

Becattini *et al* [11] suggest two slight variations of the mappings defined by Doppke: *workspace-centred mapping*; and *artefact-centred mapping*. A workspace-centred mapping is similar to the agent-centred mapping suggested by Doppke in the sense that software engineer's workspace is represented as a room. However, in a workspace-centred mapping, artefacts and tools are collected on top of desktops, and each desktop is associated with a task to be executed. Likewise, an artefact-centred mapping is similar to a product-centred mapping defines by Doppke in the sense that artefacts are mapped into rooms. However, in an artefact-centred mapping, rooms and connections correspond to the breakdown structure of the artefacts being used and developed, thus, representing composition relations amongst artefacts.

In other work, Dossick and Kaiser [28] concentrate on the support for user awareness within a PSEE. They develop a (persistent) software engineering support environment called the Columbia Hypermedia Immersion Environment (CHIME). CHIME is made up of three components:

- *Groupspaces* which provide a persistent organisation of artefacts.
- *Groupviews* which address multi-user interfaces, including awareness mechanisms, by generating rooms and connection between rooms based on the organisation of artefacts in the groupspaces.
- *Software Immersion* which creates an immersive environment from the artefacts populating the groupspaces.

In CHIME, software engineers are represented as avatars and can potentially encounter other avatars in the virtual environment. Artefacts and tools are organised in a persistent virtual space allowing software engineers to manipulate, create, modify and maintain them.

As has been shown, supporting software processes via integration with a virtual environment is not in itself new. However, such integration has mainly been an issue for the PML support environments (Process Centred Software Engineering Environments – PSEE) rather than being a feature that can be used in constructing a process model.

While integration with a virtual environment at the PSEE level is useful, it may be insufficient to address awareness and visualisation issues effectively in terms of each activity's work context. This is because although a PSEE may have the capability to proscribe a software engineer's action, it may not be able to properly capture each activity's work context. In order to overcome this limitation, there is obviously a need for a PML support. Through a PML, each activity can be both prescribed and proscribed based on its work context, hence, awareness and visualization issues can be supported accordingly. Consequently, the research for VRPML differs from the work described above by investigating exploitation of a virtual environment not at the PSEE level but at the PML enactment level.

As will be seen in Chapter 4 and Chapter 6, VRPML allows work contexts for a particular activity to be defined and later be opened as a workspace in a virtual environment when that activity is enabled using the task-centred mapping discussed above. It is, therefore, through a workspace that VRPML supports awareness and visualisation issues.

3.2 Visual Programming

The second issue of importance for the development of VRPML relates to visual programming. Visual programming can be defined as the use of graphics (e.g. graphs, icons, nodes, and arcs) supplemented with textual annotations to allow a user to specify a program. As pictures supposedly fit well into the mental representation of humans, it is believed that using visual notations may alleviate some of the difficulties inherent in writing and comprehending complex programs.

As an analogy, one may imagine a visual language as conventional road signage. Through the judicious use of graphics, drivers around the world can comprehend the same meaning of the road signs regardless of the language they speak – that is, graphics can cross language barriers. Such an example demonstrates how intuitive and powerful graphics can be if they are properly exploited.

Motivated by a number of the existing visual PMLs (e.g. FUNSOFT Nets [29], SLANG [9], Dynamic Task Nets [37], APEL [24], and EVPL [36]), a flow-based visual language paradigm seems to be a suitable choice for VRPML. Normally, flow-based visual languages are based on directed graphs. Graphs typically consist of nodes, arcs and sub-graphs. Nodes represent function or actions, arcs carry data or control-flow signals, and sub-graphs provide abstraction and modularisation. Operations in graphs follow a *firing rule* which defines the conditions under which execution of node occurs.

In the control-flow based model, a visual program consists of nodes connected by arcs carrying control-flow signals. Arcs depict the control-flow dependencies amongst connected nodes. The firing rule is based solely on the availability of the control-flow signals on the node's input arcs – that is, data availability does not play any part at all.

Conceptually, in the control-flow based model, every program can be thought of as having an instruction counter and a globally addressable memory which hold programs and data objects whose contents are updated by program instructions during execution [2, 3]. As far as a visual language associated with the control-flow model is concerned, for simplicity, it may be viewed as supporting executable flowcharts.

In the data-flow based model, a visual program consists of nodes connected by arcs carrying data. Arcs depict data dependencies amongst nodes. The firing rule is based on the availability of data on the node's input arcs, and may be *data-driven* or *demand-driven*. With a data-driven firing rule, an arc is used as a supply route to transmit data from the source node to the destination node. A destination node is executed as soon as data is available on all input arcs. With a demand-driven firing rule, an arc is used as a demand route to request data from the source node. A source node is executed only if there is a demand for its result. For either firing rule, arcs are conduits for data. In turn, data on an arc is consumed by the executing node to perform its computation (although some variations of the data-flow based model also allow an arc to retain data).

According to Agerwala and Arvind [3], the data-flow based model can be distinguished from the control-flow based model in that it has neither a globally addressable memory nor a single instruction counter. As the data-flow based model possesses no global memory, the only data available to a node for its operation is that from its inputs. In addition, because of the lack of any shared data amongst nodes, there can be no *side effects* (one node interfering with other node's data, potentially causing unexpected results).

As the data-flow firing rule depends solely on the availability of data, nodes whose data is available can potentially be executed in parallel. The sequencing of the execution of nodes, for example in terms of the assignment of runtime processes to processors, is determined solely at runtime by the runtime system. Thus, a data-flow based model supports parallelism naturally.

Apart from the control-flow or the data-flow based models, one less popular paradigm is the computational model based on both models. Here, there are two kinds of arcs with different semantics: the data-flow and the control-flow arc. The firing rule for this paradigm can be complex because it is based on the combination of both the data-flow and the control-flow signals. Furthermore, while the problem of arcs crossing each other and resulting in a cluttered view is inherent in a flow based visual language based on directed graphs, the fact that two arcs are used here means that the crossover problem can be even greater. Generally, if there are too many arc crossovers, the overall program understanding may be compromised.

Therefore, the option of combining the control and data-flow models will not be considered as the computational model for VRPML.

Having disregarded the option of adopting both the data-flow and the control-flow as the computational model for VRPML, there are now only two choices of the computational model: the control-flow model; or the data-flow based model. The selection of the computational model for VRPML will be discussed next by taking into accounts the need of a software process.

In order to support modelling and enacting of software processes, there is a need for VRPML to support cyclic behaviour. This need can be seen, for example, in the case where software design fails its review. In the case of the control-flow model, only one control-flow signal is needed to enable the previous node regardless of the number of data items required for that node. However, in the case of the data-flow based model, to enable the previous node requires all the data for that node to be available. Thus, while both the control-flow and the data-flow based model can address this need in a straightforward manner, it seems somewhat easier in terms of implementation to use the control-flow rather than the data-flow model

In addition to being able to support cyclic behaviour, there is also a need for VRPML to facilitate reasoning about its execution semantics, hence making enactment of a software process traceable. It is useful to be able to trace the enactment of the software process to facilitate process awareness and process understanding which can lead to improved process support. Because the ordering and enactment of tasks in the data-flow based model strictly depends on the availability of data at runtime, enactment of a software process may be non-deterministic. Hence, it is difficult to reason about and to trace enactment in the data-flow model compared to the control-flow model.

Because some activities in a software process need to be undertaken by more than one software engineer, there is also a need for VRPML to support shared artefacts. For example, in the case of an activity such as modify code, it may be that more than one software engineer is assigned to change the same source code at the same time. Although both the control-flow and the data-flow based models can address this need, it seems preferable to use the control-flow rather than the data-flow model. Because the data-flow model inherently avoids the problem of side effects –

that is, one node interfering with other node's data, there is a need for copies of data to be replicated across nodes to enable shared data. In fact, this need may put an extra burden for process engineers who construct the process model.

Finally, although the data-flow based model is helpful in the sense that parallelism can be achieved automatically, such a feature may not be a major benefit for enacting software processes in VRPML. As far as enactment of software processes are concerned, because software engineers are the "processors" which perform the computation, it is desirable to have their assignments under human control (e.g. under the discretion of the project manager). This is because software engineers have different skills which need to be considered before task assignments can be made. Thus, in this respect, the data-flow based model has no clear advantage over the control-flow based model.

While the data-flow based model has some merits, it is the control-flow based model that has been chosen for VRPML. This choice will be reflected in the design of VRPML in the next chapter.

3.3 Supporting Dynamic Allocation of Resources

As discussed in Chapter 2, the current approach through which the dynamic allocation of resources can be achieved in a PML is via enabling evolution of the process model while enactment is taking place. While enabling evolution is a necessary feature of a PML, extra overhead may be introduced because of the need to enable steps in the meta-process to ensure no *ad hoc* changes and no side effects might be introduced in the process model. Therefore, because of this extra overhead, it seems unjustified to exploit on-the-fly process evolution just to achieve dynamic allocation of resources.

An alternative way of addressing the support for dynamic allocation of resources is to define a suitable enactment model which is able to cope with the required dynamic behaviour. A candidate is an enactment model which exploits a resource exception handling mechanism. Briefly, whenever the runtime system fails to acquire resources for a particular activity, a resource exception will be thrown causing the enactment of that activity to be suspended. In turn, another activity is automatically created and assigned to the process engineer to fix the resource

exception. Only after the resource exception is properly fixed can enactment of that activity is resumed. This enactment model will be further discussed in Chapter 4 in details after the description of VRPML syntax and semantic are developed.

3.4 Summary

This chapter started with a brief survey of the related work which suggests that it is useful to exploits a virtual environment as a way to address the human dimension issues at the PML enactment level. Then, a general discussion on visual programming as well as the justification for the adopting the control-flow model for VRPML was provided. Finally, a brief remark on the limitation of the current approaches for achieving dynamic allocation of resources was given, and a novel alternative approach was identified for VRPML.

Building on the material on this chapter and the characteristics of PMLs given in Chapter 2, the next chapter looks into the actual syntax and semantic of VRPML. Additionally, issues relating to the enactment model are developed further.

Chapter 4 – The VRPML Notation

The previous three chapters gave a broad overview of background research for VRPML. In particular, they discussed the current state-of-the-art with respect to PMLs, provided justification and requirements for the development of VRPML as well as outlined issues related to visual programming together with rationalisation of design choices which VRPML is based upon. The remainder of this thesis will focus on the design of VRPML, modelling of the case studies, and prototype implementation and evaluation.

This chapter starts by giving an overview of VRPML notation by revisiting the partial solution of the ISPW-6 problem [48] originally introduced in Chapter 1. Subsequently, the detailed syntax and semantics of VRPML are described. Next, the enactment model is discussed illustrating how enactment and the support for the dynamic allocation of resources can be achieved.

4.1 Overview

VRPML is a control-flow based visual language for supporting modelling and enacting of software processes in a virtual environment. In VRPML, software processes are generically modelled in that resources (in terms of software engineers, artefacts and tools) are allowed to be allocated dynamically based on the specific need of a particular project.

Software processes are written in VRPML as graphs, by interconnecting nodes from top to bottom using arcs carrying control-flow signals. In terms of its structure, VRPML graphs are cyclic. As an illustration for VRPML syntax, Figure 4-1 revisits the partial solution of the ISPW-6 problem given in Chapter 1 – the overall explanation of the VRPML solution of the ISPW-6 problem will not be fully discussed until the next chapter.

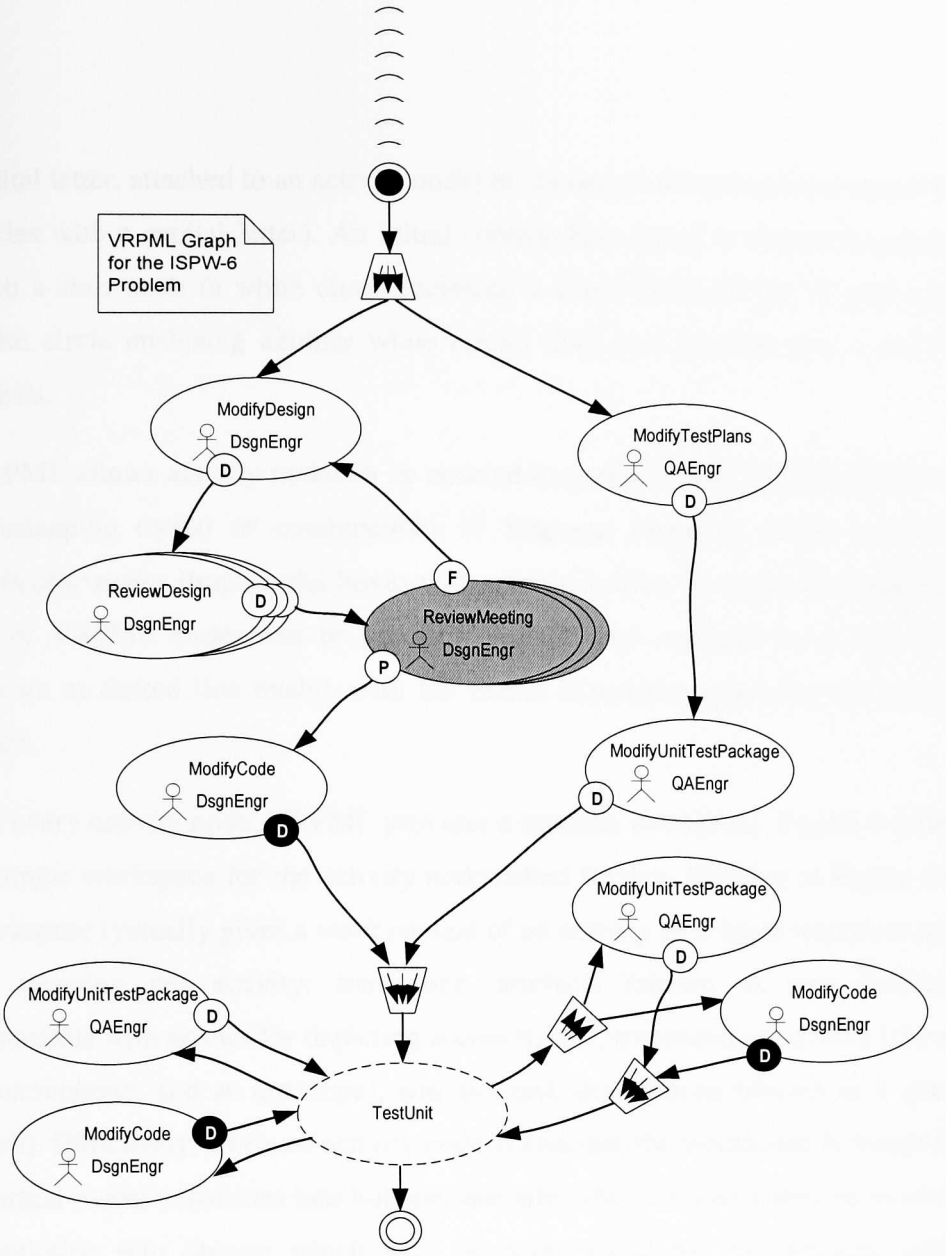


Figure 4-1 Partial Solution of the ISPW-6 Problem Revisited

Similar to JIL [63] and Little JIL [66], software processes in VRPML are described using process steps, which represent the most atomic representation of a software process (i.e. the actual task that software engineers are expected to perform). These process steps are represented as nodes, called *activity nodes* (shown as small ovals with stick figures). VRPML supports a number of different kinds of activity nodes – some examples are in Figure 4-1, but are discussed further in Section 4.2.3.

The firing of activity nodes is controlled by the arrival of a control-flow signal. Control-flow signals may be generated at the completion of a node, often from special completion events called *transitions* (shown as small white circles with a

capital letter, attached to an activity node) or *decomposable transitions* (small black circles with a capital letter). An initial control-flow signal is always be generated from a *start node* (a white circle enclosing a small black circle). A *stop node* (a white circle enclosing another white circle) does not generate any control-flow signals.

VRPML allows activity nodes to be enacted in parallel using multi-instance nodes (overlapping ovals) or combinations of language elements called *merger* and *replicator nodes* (trapezoidal boxes with arrows inside). To improve readability, a set of VRPML nodes can be grouped together and replaced by a *macro node* (shown as dotted line ovals), with the macro expansion appearing on a separate graph.

For every activity node, VRPML provides a separate *workspace*. Figure 4-2 depicts a sample workspace for the activity node called Review Meeting in Figure 4-1. A workspace typically gives a *work context* of an activity as it hosts resources needed for enacting the activity: transitions, artefacts (shown as two overlapping documents with arrows for depicting access rights), communication tools (shown as a microphone, and an envelope), and any task descriptions (shown as a question mark). Effectively, when an activity node is enacted, the workspace is mapped into a virtual room, transitions into buttons, and artefacts, communication tools and task description into objects which may be manipulated by the assigned software engineer to complete the particular task at hand.

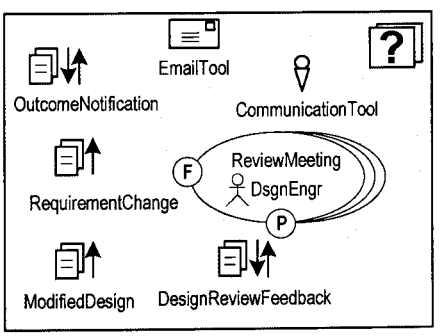


Figure 4-2 Sample workspace for the Review Meeting in Figure 4-1

Having defined the general flavour of VRPML, the syntax and semantic of VRPML can now be explained in details.

4.2 Syntax and Semantic of VRPML notation

This section discusses the syntax and semantic of VRPML notation as well as their main justification. Where appropriate, an example will be provided from the VRPML solution of the ISPW-6 problem to demonstrate how the notation should be used for supporting modelling and enacting of software processes.

4.2.1 Start, Stop and Re-enabled Nodes

As VRPML adopts the control-flow model, it is necessary to be able to generate and propagate control-flow signals. In addition to the use of transitions (described below), the way VRPML provides and handles a control-flow signal is through a start node, a stop node and a re-enabled node. The notations for start nodes, stop nodes and re-enabled nodes are given in Figure 4-3.

Figure 4-3 Start, Stop and Re-enabled Nodes

A start node generates an initial control-flow signal when the VRPML graph is first enacted. A stop node serves as an exit point for control-flow signals. A re-enabled node behaves like a stop node allowing the control-flow signal to exit and permitting the same control-flow signal to re-enable its own parent's activity node – their use will be clarified when decomposable transitions are explained below.

4.2.2 Arcs

In VRPML, arcs are conduits for control-flow signals. Arcs also depict dependencies amongst activity node and their proper sequencing. In VRPML, arcs are unidirectional and only one control-flow signal may flow on a particular arc at any particular time. The obvious notation for an arc is given in Figure 4-4.

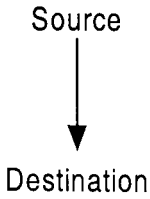


Figure 4-4 Arc

4.2.3 Activity Nodes and Workspaces

Process steps are used as the main abstraction for software processes in VRPML, as in JIL and Little JIL. They are represented as activity nodes and workspaces. The concepts of activity nodes and workspaces are analogous to package interfaces and package bodies in the ADA programming language that is, activity nodes and workspace come in pairs. For every activity node, a workspace must be defined.

In general, an activity node is a parameterised node, accepting a role assignment as a parameter which may be used to allocate a specific software engineer to the task. For a particular activity node, a workspace hosts transitions, and artefacts and communication tools for completing the activity. In addition, for every workspace, a task description can be optionally specified to informally describe the activity in natural language and be made available during enactment to guide the software engineer. In VRPML, the concept of workspaces can be seen as a step toward supporting process visualisation.

There are three kinds of activity nodes supported by VRPML. They are a *general-purpose activity node*, a *multi-instance activity node* and a *meeting activity node*. The visual representation of activity nodes and their workspaces is given in Figure 4-5.

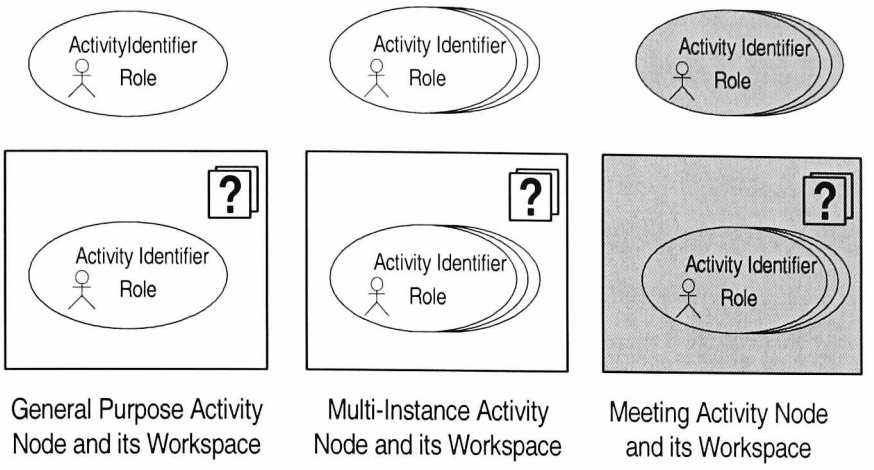


Figure 4-5 Activity Nodes and Their Workspaces

A general-purpose activity node represents the most atomic part of software processes which a software engineer is expected to perform. A multi-instance activity node, a concept borrowed from APEL [24], is a collection of identical process steps performed by more than one person. A meeting activity node allows multiple software engineers to meet virtually to make some collective decision. Although not exemplified in their icons above, both multi-instance activity nodes and meeting activity nodes have an associated depth, indicating the actual number of engineers involved (and also the number of identical activities in the case of multi-instance activity). Depths of multi-instance activity and meeting activity nodes can be specified during instantiation or dynamically.

As depicted in Figure 4-5, workspaces for different kinds of activity nodes are unique. The reason for having a unique workspace is to support a sense of process awareness during process enactment. For instance, software engineers are able to distinguish whether the process steps that they are undertaking also concurrently involve other software engineers – the case for multi-instance activities. Such awareness should encourage inter-person communications, which is seen as one of the important aspects of supporting collaborative work [68].

4.2.4 Transitions

When a process step specified by an activity node is undertaken in its workspace, there is obviously a need to be able to indicate its completion (success or failure) or

cancellation. The way VRPML support such events is through *transitions*. Effectively, transitions give a software engineer some flexibility and choice of control over what happen next. This is an important characteristic, which can also be seen in other PMLs, particularly Little JIL. In term of semantics, a transition is similar to a button (in the engineer's virtual room), which responds to a click-event. On pressing a transition, a single control-flow signal is generated.

In the case where certain post-conditions would have to be satisfied or some automated steps would need to be performed before allowing the completion or cancellation of a process step, VRPML allows transitions to be decomposable. *Decomposable transitions* enable automation scripts or sub-graphs to be specified (and executed if selected). An example of a transition labelled D (representing a 'done' transition) and a decomposable transition labelled A (representing an 'abort' transition) is given in Figure 4-6.

Figure 4-6 Example of Transition and Decomposable Transition

The re-enabled node introduced earlier behaves like a stop node allowing a control-flow signal to exit through in a decomposable transition and permitting the same control-flow signal to re-enable its parent's activity node. To illustrate the behaviour of a re-enabled node, Figure 4-7 depicts a sub-graph representing the decomposable transition labelled D (for Done) for the activity node Modify Code in Figure 4-1. If the transition representing Redo (labelled R) in the activity node Check Compilation in Figure 4-7 is pressed by the assigned software engineer, a control-flow signal will be generated and will re-enable its parent activity node Modify Code. On the other hand, if the transition representing Done (labelled D) in the activity node Check Compilation is pressed by the assigned software engineer, that is, in the case where codes are successfully compiled, a control-flow signal will be generated to flow out of the stop node and back to the main VRPML graph in Figure 4-7 (i.e. towards the merger node).

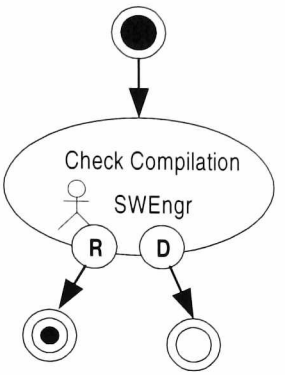


Figure 4-7 Example Usage of a Re-enabled Node

ess given to the assigned software engineer for transitions is handled ly for different kinds of activity nodes. In a general-purpose activity node, ned software engineer has sole access to the transitions in that node. In a stance activity node, each assigned software engineer also has sole access instance of the transitions but only one control-flow signal will be d eventually from the node. For this signal to be generated, all of the multi- transitions must be completed. Control-flows generated for each of these ns can be thought of as passing through a merger node (described below).

eting activity node, access to transitions is only given to a designated engineer who is expected to moderate the meeting. By convention, the engineer assigned to the top general-purpose activity node in a meeting node is assumed to be the moderator. Meeting activity nodes can also be support process steps involving multiple software engineers, but access to ns will be centrally controlled by the moderator.

Macro Nodes

ort modularisation of its graphs, VRPML provides a macro facility. A *ode* can group one or more nodes together and replace them with a single reduce the graph complexity. Semantically, a macro node behaves like a macro in the C programming language. When a control-flow signal ers a macro node, the enacted VRPML graph is rewritten to include the ntained in that macro. In this way, a macro may be used in different parts

of VRPML graphs. The visual representation of a macro node is given in Figure 4-8.

Figure 4-8 Macro Node

To illustrate the usage of a macro node, Figure 4-9 shows the expansion of the macro node called Test Unit in Figure 4-1. Transitions on the Test Analysis node have been labelled as: B (for Both Feedback); C (for Code Feedback); T (for Test Feedback); and P (for Passed).

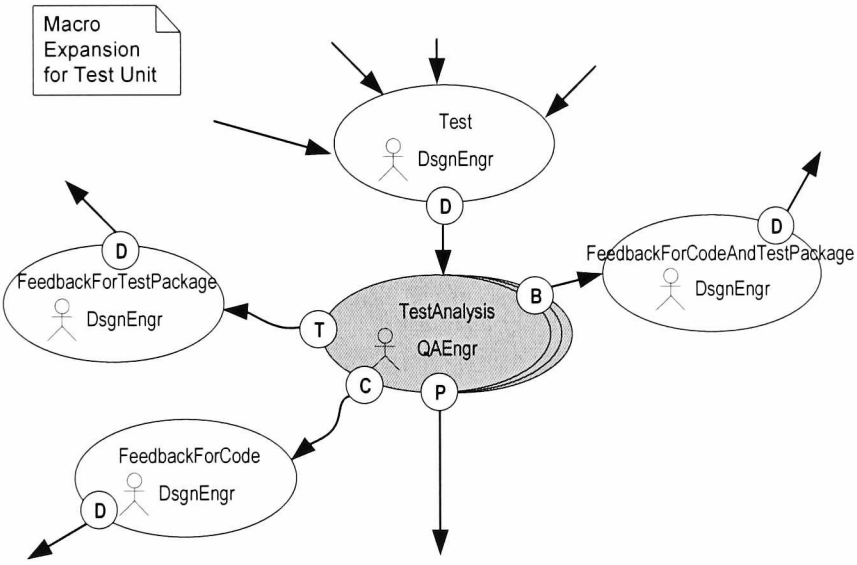


Figure 4-9 Macro Expansion for Test Unit (see Figure 4-1)

4.2.6 Replicator and Merger Nodes

Since software processes in reality are never performed in a sequential manner, a PML needs to support process steps running in parallel. As well as the use of multi-instance nodes, in VRPML this parallelism support is enabled using *replicator* and *merger nodes*. Replicator nodes accept a control-flow signal and replicate it as many times as specified by arcs leaving it. Merger nodes accept many control-flow

signals and convert them into a single signal. In doing so, a merger node blocks execution until it receives the control-flow signals on all of its input arcs. The visual representation of replicator and merger nodes is given in Figure 4-10.

Figure 4-10 Replicator and Merger Nodes

4.2.7 Artefacts

Artefact icons provide access to the actual artefacts and tools needed to complete the particular process step, and can only be defined inside a workspace. An artefact has two formal parameters. These parameters are associated with the actual location of the artefact (e.g. a particular source file) and the corresponding tool and its actual location (e.g. the compiler to use). Typically, these parameters are set either during instantiation or dynamically during enactment. Artefacts have 5 modes of access: *read-only*; *read and write*; *write-only*; *local*; and *exclusive*. The default mode of access for artefacts is read and write. With the exception of exclusive and local, other modes of access are similar to those in Little JIL. Exclusive access mode is needed in the case of multi-instance activity nodes and meeting activity nodes where artefacts are by default shared amongst assigned engineers. By using the exclusive access mode, certain shared artefacts can be made exclusively accessible to a particular software engineer. Local access mode allows the definition of artefacts local to a particular activity node (i.e. similar to "local variable" in a general programming language). The visual representation of artefacts and their modes of access is given in Figure 4-11.

Figure 4-11 Artefacts and Their Access Rights

4.2.8 Communication Tools

Communication tools support interactions amongst software engineers, which is an important element when geographically and temporally distributed software engineering teams are involved in the processes. Synchronous (e.g. teleconferencing programs or chat-like programs) and asynchronous (e.g. email or messaging programs) tools are provided as icons and have one formal parameter. This parameter, which may be set during instantiation or dynamically, relates to the actual tool and its location. Just as with artefacts, communication tools must be defined inside a workspace. One way of providing user awareness support is to use videoconferencing-like systems as the main synchronous communication tool. The visual notation for synchronous and asynchronous communication tools is given in Figure 4-12.

Figure 4-12 Synchronous and Asynchronous Communication Tools

4.2.9 Comments

Finally, like other PMLs, VRPML also provides support for specifying comments. In general, comments allow extra documentation, and provide some justification pertaining to the chosen approach in the source code. In VRPML, comments can be placed anywhere. The visual representation of a comment is given in Figure 4-13.

Figure 4-13 Comments

4.3 Enactment Model

An issue of paramount importance relating to VRPML's enactment model is resources and their allocation. In VRPML, resources include roles assignment, artefacts and tools (including synchronous and asynchronous communication tools) in a workspace as well as the depths of multi-instance activity nodes and meeting activity nodes. Since software processes are highly dynamic, resources can rarely be specified completely during instantiation (discussed in Chapter 2), and require dynamic support during enactment. VRPML exploits a resource exception handling mechanism to provide this support.

If for any reason a graph enactment fails to acquire its resources to enable an activity, enactment will be blocked until such resources are made available. In this way, VRPML resource exception handling mechanism is similar to blocking primitives (e.g. in, read) in Linda [33].

As software processes are typically long-lived, blocking removes the need to recover from the effect of previously completed process steps, thus, allowing online rectification of an exception. Ideally, such a feature allows enactment of a process model even if resources have not yet been assigned.

Figure 4-14 depicts the enactment model for an activity node in VRPML expressed in terms of a state transition diagram. The behaviour of the runtime systems supporting such an enactment model can be thought of as consisting of a single producer (VRPML interpreter) and multiple consumers (engineer's runtime support system) communicating using a shared tuple space.

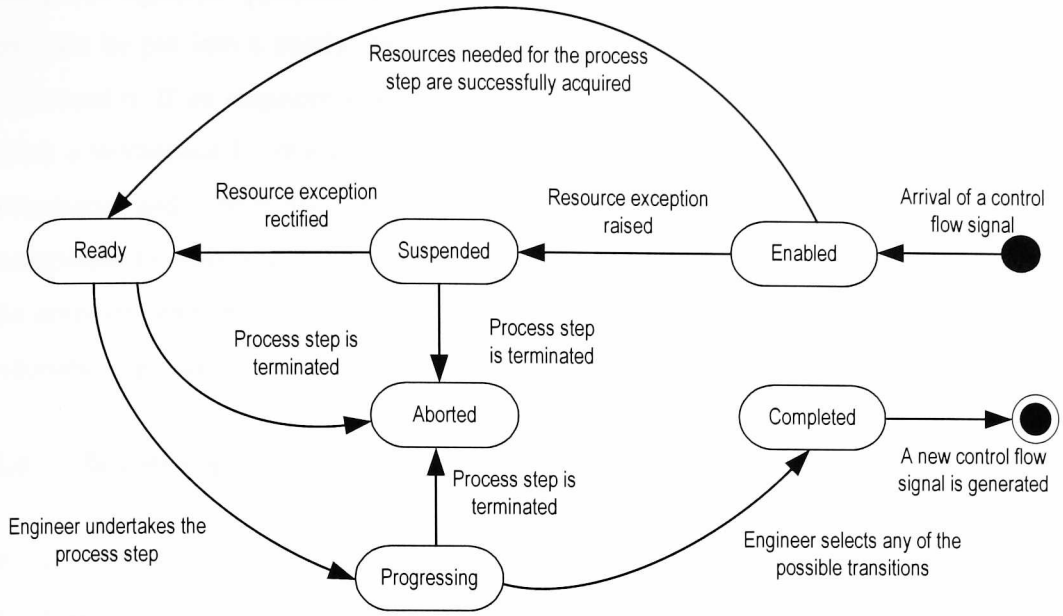


Figure 4-14 VRPML Enactment Model

Upon the arrival of a control-flow signal, an activity node will be in an **enabled** state. This is the case where the VRPML interpreter attempts to acquire resources (in terms of role assignments, artefacts, tools as well as depths of activity nodes) that the activity node needs. If resources are successfully acquired, the VRPML interpreter then "produces" the process step corresponding to that activity node in the tuple space. The engineer's runtime support system then "consumes" the process step putting it into a **ready** state. Ideally, in this state, the process step is made available in the to-do-list of the assigned software engineer. If for any reason, VRPML interpreter fails to acquire resources it needs, a resource exception will be thrown putting the enactment of that particular process step in the VRPML graph into a **suspended** state. In this state, VRPML interpreter automatically produces a process step in the tuple space for the administrator (in this case, it may be the project manager or "process engineer") to rectify the resource exception or completely terminate the process step (putting it in an **aborted** state). If a process step is terminated, the administrator may optionally terminate the overall enactment of the particular VRPML graph in question or manually re-enact connecting and enclosing nodes (e.g. in a decomposable transition) by providing the necessary control-flow signals that they need to fire. If the resource exception is rectified, enactment of the particular VRPML graph can continue allowing VRPML

interpreter again to "produce" the process step in the tuple space. This process step can then be put into a **ready** state once the engineer's runtime support system has consumed it. If an engineer selects that particular process step (in the **progressing** state), a workspace for that process step will appear as a virtual room with artefacts, transitions and communication tools as objects which software engineer can manipulate to complete the task. The process step is in the **completed** state when the software engineer selects any one of the possible transitions regardless of its outcome (e.g. passed, failed, done, or aborted).

4.4 Summary

In summary, this chapter discussed the syntax and semantics of VRPML intended to support modelling and enacting of software processes in a virtual environment. Toward the end, this chapter outlined the VRPML enactment model to illustrate how enactment and dynamic allocation of resources can be achieved.

The next chapter presents the detailed account of the VRPML solution of the ISPW-6 problem. Additional experience with VRPML to support the modelling of the waterfall software development model is also summarised.

Chapter 5 – Modelling of Two Case Studies Using VRPML

The previous chapter described the VRPML notation in terms of its syntax and semantics by outlining the partial solution of the ISPW-6 problem. Building from this material, this chapter presents VRPML solutions of two case study problems: the complete ISPW-6 problem; and the waterfall software development model. The main aim of this chapter is, therefore, to investigate whether VRPML provides a sufficiently rich notation to support the modelling and enacting of software processes.

5.1 Experience with VRPML

In order to explore the expressiveness of the VRPML notation, there is obviously a need to apply VRPML to case study problems. One well-known case study problem for facilitating evaluation of a PML is the ISPW-6 problem [48]. The ISPW-6 problem represents a standard benchmark for determining whether a new PML can express a real software engineering process. Furthermore, through use of this standard benchmark problem, an objective comparison of VRPML with other PMLs can be made.

Since the ISPW-6 problem is specific to the software change request process, it may be insufficient to evaluate VRPML completely. Therefore, a more general case study problem is also needed, particularly involving the software processes for a complete software development model. These processes must be explicit and well-defined in terms of their inputs and outputs. Arguably, if one could use an existing definition of a development model in which activities and their inputs and outputs have already been well-defined, more effort can be concentrated on the modelling and enacting issues and less on defining the stages (and activities). The waterfall software development model seems to fit well into this category.

In general, the waterfall software development model relies on a stage-at-a-time approach, and consists of a number of well-defined stages. Most of these stages have fixed constraints, and culminate in formal agreed milestones, many of which

involve documentation. Thus, the waterfall software development model is a convenient choice as another case study for VRPML. In the next section, a discussion of the two case studies will be offered along with their solution expressed in VRPML.

5.2 The ISPW-6 Problem

The ISPW-6 problem [48] concerns a software requirement change request for an existing software component, occurring either towards the end of the development phase or during maintenance phase of the software lifecycle. The ISPW-6 problem starts in response to a software requirement change request, with the project manager scheduling and assigning various engineering tasks in the process. These tasks include: Schedule and Assign Tasks; Modify Design; Review Design; Modify Code; Modify Test Plans; Modify Unit Test Package; and Test Unit. During the enactment of these tasks, the project manager is also required to monitor their progress. Figure 5-1 depicts the flow of tasks in the ISPW-6 problem.

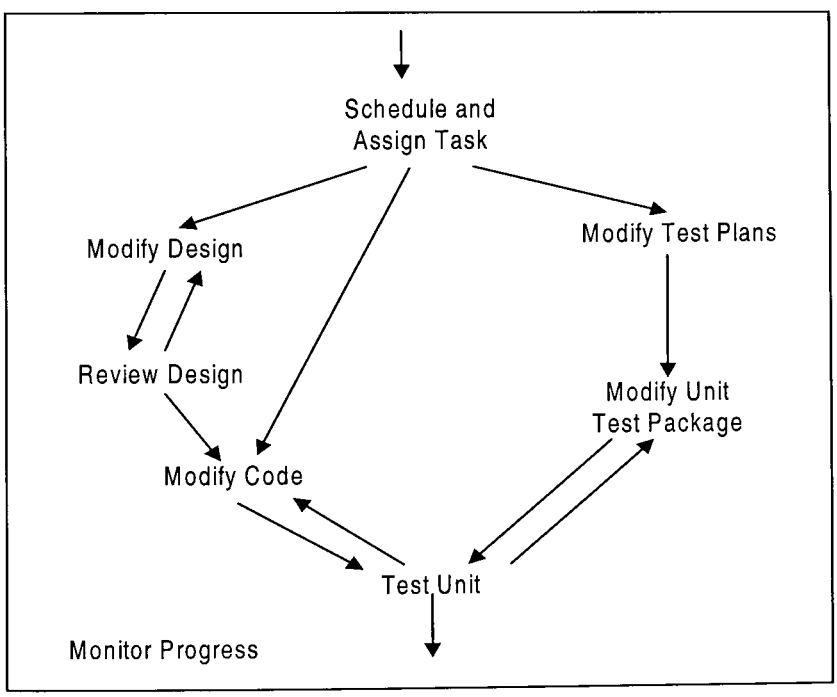


Figure 5-1 Flow of tasks in the ISPW-6 Problem

In each of the tasks, the ISPW-6 problem defines restrictions in terms of the ordering, the input and output artefacts, and the role responsible for each activity,

as well as conditions for each activity initiation and termination. Table 5-1 summarises the role responsibility, inputs and outputs as well as constraints defined for each of the tasks.

Schedule And Assign Tasks	Responsibility: Project Manager
	Inputs: Requirement Change, Authorisation, Project Plans
	Outputs: Updated Project Plans, Notification of Task Assignments and Schedule Dates, Requirement Change
	Constraints: - Begins as soon as authorisation is given - Ends when outputs have been provided
Modify Design	Responsibility: Design Engineer
	Inputs: Requirement Change, Current Design, Design Review Feedback
	Outputs: Modified Design
	Constraints: - Can begin as soon as the task been assigned - Subsequent iteration can begin if design is not approved by the Review Design - Ends when outputs have been provided
Review Design	Responsibility: Design Review Team
	Inputs: Requirement Change, Modified Design
	Outputs: Design Review Feedback, Approved Modified Design, Outcome Notification
	Constraints: - Begins on schedule provided the modified design is available at the time - Ends when outputs have been provided
Modify Code	Responsibility: Design Engineer
	Inputs: Requirement Change, Modified Design, Current Source Code, Feedback Regarding Code
	Outputs: Modified Source Code, Object Code
	Constraints: - Can begin as soon as the task has been assigned even if Modify Design has not begun (discretion) - Ends when clean compilations are achieved, outputs have been provided and design is approved - Subsequent iteration can begin if required when test unit has completed
Modify Test Plans	Responsibility: QA Engineer
	Inputs: Requirement Change, Current Test Plans
	Outputs: Modified Test Plans
	Constraints: - Can begin as soon as the task has been assigned - Ends when outputs have been provided
Modify Unit Test Package	Responsibility: QA Engineer
	Inputs: Requirement Change, Modified Test Plans, Current Unit Test Package, Modified Design, Source Code, Feedback Regarding Test Package
	Outputs: Modified Unit Test Package
	Constraints: - Can begin as soon as Modify Test Plans has completed - Subsequent iteration can begin if required as Test Unit has completed - Ends when outputs have been provided
Test Unit	Responsibility: Design Engineer, QA Engineer
	Inputs: Requirement Change, Object Code, Unit Test Package
	Outputs: Test Results, Feedback Regarding Code, Feedback Regarding Test Package, Notification of Successful Testing
	Constraints: - Can begin as soon as both Object Code and Unit Test Package are available - Ends when outputs have been provided
Monitor Progress	Responsibility: Project Manager
	Inputs: Requirement Change, Notification of Completion (from all tasks), Current Project Plans, Outcome Notification, Notification of Successful Testing, Decision Regarding Cancellation
	Outputs: Updated Project Plans, Notification of Revised Task, Cancel Recommendation
	Constraints: - Persists throughout the duration of the process - Ends when Test Unit has been successfully completed or cancellation of the whole ISPW process

Table 5-1 Summary of the ISPW-6 Problem

Furthermore, some tasks defined in the ISPW-6 problem also involve a collective decision making process: Review Design and Test Unit. In Review Design, there are three possible outcomes of the review process:

- Unconditional approval – in this case, the design is totally approved and incorporated into the software design document.
- Minor changes recommended – in this case, minor changes are required and feedback is provided to the design engineer. The re-review is expected to be perfunctory.
- Major changes recommended – in this case, major changes are required and feedback is provided to the design engineer.

In the Test Unit, there are four outcomes of the testing process depending on the test coverage attained by running the unit test package. The possible outcomes are:

- The Test Unit is successful as 90% test coverage has been achieved (assuming an automated coverage analyser is employed). In this case, the ISPW-6 problem completes.
- The source code needs to be modified further. In this case, feedback regarding the code must be furnished and the task Modify Code has to be restarted.
- The unit test package needs to be modified further. In this case, feedback regarding the test package must be furnished and the task Modify Unit Test Package has to be restarted.
- Both source code and unit test package need to be modified further. In this case, feedback regarding both the code and test package must be furnished. In addition, tasks Modify Code and Modify Unit Test Package have to be restarted.

Having described the ISPW-6 problem in detail, the solution expressed in VRPML can now be presented.

5.2.1 VRPML Solution of the ISPW-6 Problem

From the above discussion, one can observe that there are essentially two different process models within the ISPW-6 problem. In the first model, the Modify Code is

started after Modify Design and Review Design. In the second model, Modify Code is started in parallel with Modify Design and Review Design upon the discretion of the project manager (i.e. under the assumption that the affected source code is known in advance). In order to represent both models, VRPML employs two separate graphs shown in Figures 5-2 and 5-3 respectively.

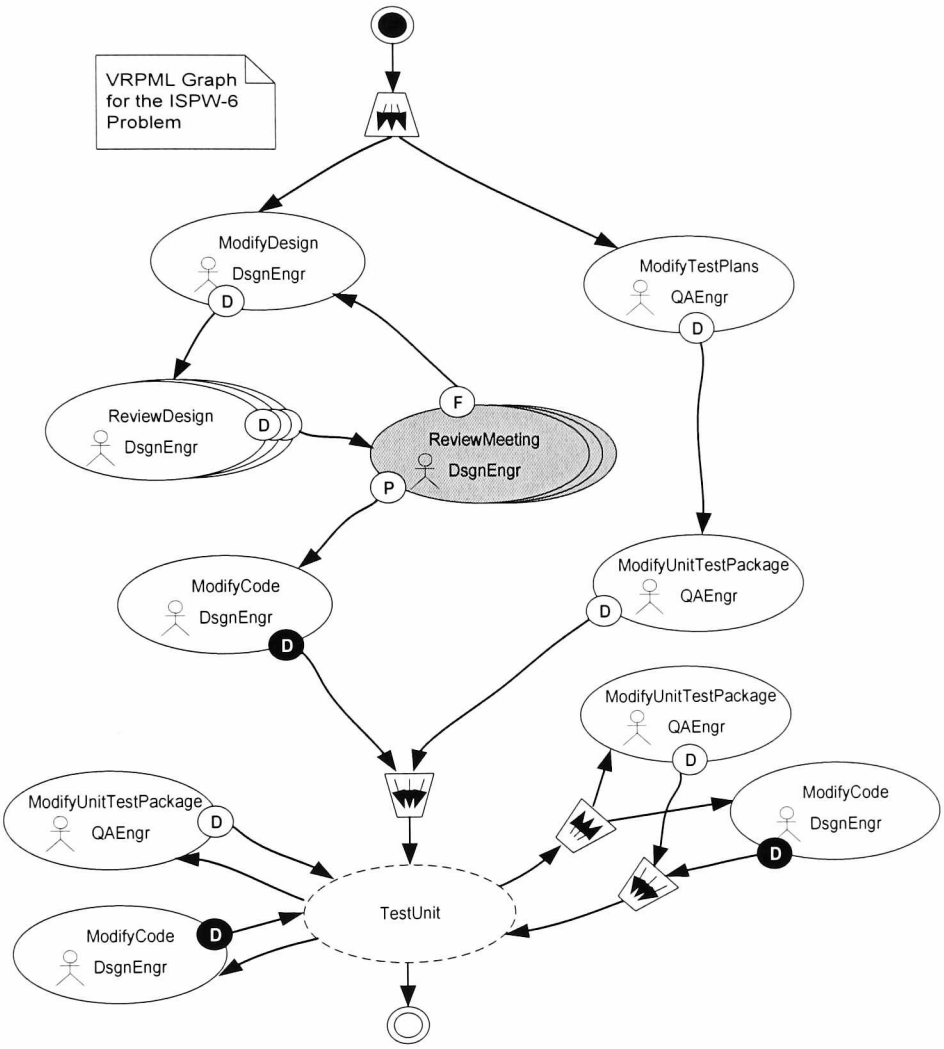


Figure 5-2 VRPML Graph for the ISPW-6 Problem

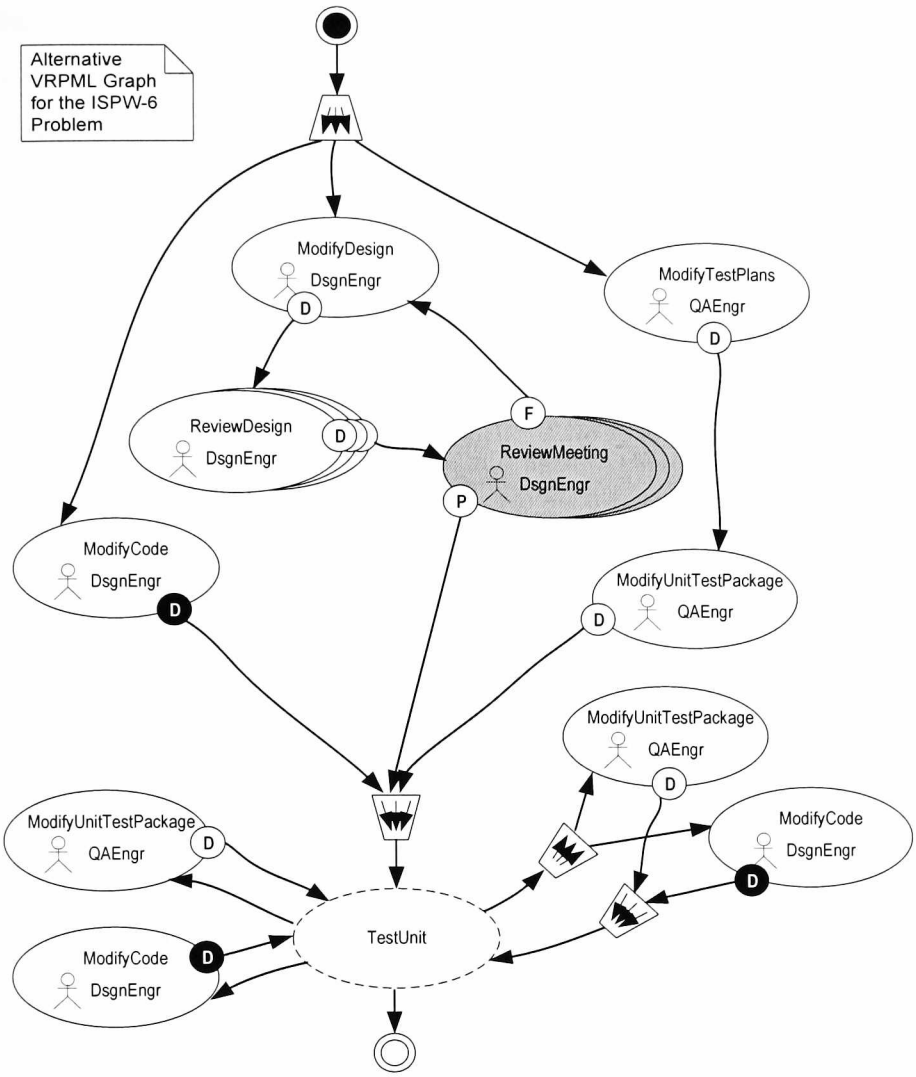


Figure 5-3 Alternative VRPML Graph for the ISPW-6 Problem

Although the VRPML graphs in Figures 5-2 and 5-3 differ in terms of the ordering and sequencing of Modify Code, the activity nodes and their workspace definitions are identical. Thus, for brevity sake, only the VRPML graph in Figure 5-2 will be developed further below.

At a glance, it might appear that the tasks Schedule and Assign Tasks and Monitor Progress in Figure 5-1, are not modelled as part of the solution. As discussed in Chapter 4, scheduling and assignment of tasks in VRPML can be performed dynamically, that is, by relying on the resource exception handling mechanism that is part of the enactment model. In fact, scheduling and assignment of tasks also includes the fact that the project manager needs to choose between the two VRPML

graphs given earlier. Additionally, the rectification of the resource exception supported by VRPML can be seen as a way to monitor progress dynamically as enactment commences. Consequently, the fact that Schedule and Assignment of Tasks and Monitor Progress are not modelled is not a major issue.

Apart from the above, most of the tasks identified in the ISPW-6 problem appear directly as activity nodes in the VRPML graph with the exception of Review Design and Test Unit. Review Design is actually broken down into two activities in the VRPML graph:

i. Review Design

ii. Review Meeting

The reason for breaking down Review Design into sub-activities is that the review process often involves two activities relating to the review of the design and the collective decision making process. In fact, Review Design and Review Meeting can be thought of as the post-condition checks on Modify Design. This is due to the fact that the actual completion of Modify Design is decided by both Review Design and Review Meeting.

Test Unit is broken down into five activities grouped into a macro node called Test Unit (described below) involving:

i. Test

ii. Test Analysis

iii. Feedback for Code

iv. Feedback for Test Package

v. Feedback for Code and Test Package

Similar to Review Design, the main rationale for breaking down Test Unit into five activities is that, apart from testing, Test Unit also involves a collective decision making process on the test outcome as well as giving the appropriate feedback on the test results.

In terms of enactment, when the VRPML graph in Figure 5-2 is first enacted, a control-flow signal is generated by the start node. This control-flow signal is then replicated by the replicator node to enable the Modify Design and the Modify Test Plans activity nodes. Upon the arrival of the control-flow signals for both activity nodes, the VRPML interpreter attempts to acquire the resources (in terms of the artefacts, roles and engineer's assignment) for both activities. If resources have either not yet been assigned or have been assigned but are not available, a resource exception will be thrown, and the enactment of the activity whose resource exception is thrown will be suspended. In turn, the VRPML interpreter automatically creates a task for the process engineer to fix the resource exception or terminate the overall enactment. Once resources have been assigned and successfully acquired, Modify Design and Modify Test Plans can now appear in the to-do-list of the assigned software engineer. When the assigned software engineer chooses to undertake the task, the workspace of the task will be opened in its virtual environment. To allow completion or cancellation of a particular task, the software engineer may select from any one of the given transitions which appear as objects in that virtual environment. In turn, the selected transition will automatically generate the appropriate control-flow signals to support further enactment.

As the enactment of the VRPML graph in Figure 5-2 is relatively straightforward, it will not be traced further. However, there are a number of issues relating to enactment which are worth discussing. Two issues relate to the enactment of the multi-instance activity node Review Design and the meeting node Review Meeting. Firstly, as discussed in Chapter 4, the depths of the multi-instance activity node and the meeting activity node correspond to how many software engineers are involved. These depths can be specified either before or dynamically during enactment. This feature again enables VRPML to support the dynamic creation of tasks according to the (dynamic) needs of a particular project.

Secondly, in the case of Review Design, a control-flow signal will only be generated if all of the assigned engineers have completed the design, that is, all of them have already selected the Done transition. Effectively, being a multi-instance activity, Review Design can be viewed as a combination of general purpose activity nodes going through a merger node. As an illustration, Figure 5-4 shows the sample equivalent graph for Review Design assigned to three engineers (Depth = 3).

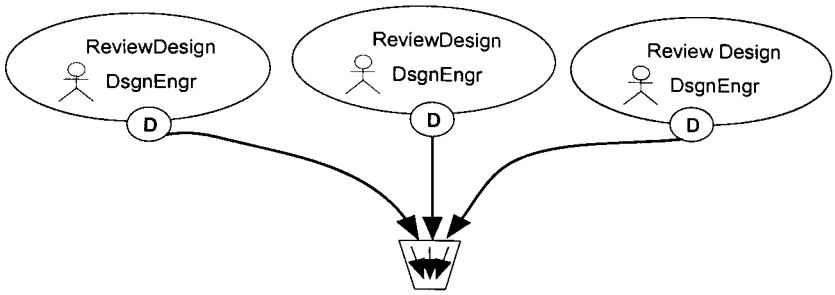


Figure 5-4 Sample Equivalent Graph for Review Design with Depth = 3

In the case of Review Meeting, a control-flow signal will always be generated when the moderator (by default, the assigned design engineer specified on top of the meeting activity node) selects any of the possible transitions, regardless of how many engineers were assigned to the meeting. Furthermore, because a meeting activity node also supports shared artefacts as a multi-instance activity node, it is possible to combine Review Design and Review Meeting into a single meeting activity node. However, the access to the Done transition in such a case will be given solely to the moderator.

The third enactment issue concerns the decomposable transition representing Done (labelled D) for the activity node called Modify Code. When the decomposable transition D is selected, a control-flow signal will be generated to enact the sub-graph associated with that transition, given in Figure 5-5.

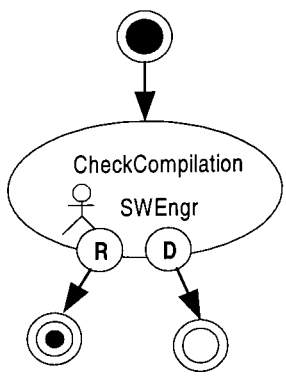


Figure 5-5 Sub-graph for the Transition Done in Modify Code

In effect, the purpose of allowing sub-graphs to be specified for a decomposable transition is to ensure that any post-conditions for its parent activity node are

satisfied before allowing further enactment. For example, in this case, the post-condition of the activity node Modify Code is that the modified code must compile successfully. Obviously, if the modified code does not compile successfully, the assigned software engineer can select the transition representing Redo (labelled R) in the activity node Check Compilation in Figure 5-5. As a result, a control-flow signal will be generated and will re-enable the transition's parent activity node Modify Code through a re-enabled node.

Finally, the fourth issue relates to enactment of the macro node called Test Unit. When a control-flow signal encounters the Test Unit macro node, the enacted VRPML graph is rewritten to include the graph contained in that macro (shown in Figure 5-6). As described earlier, the macro Test Unit consists of five activities: Test; Test Analysis; Feedback for Code; Feedback for Test Package; and Feedback for Code and Test Package.

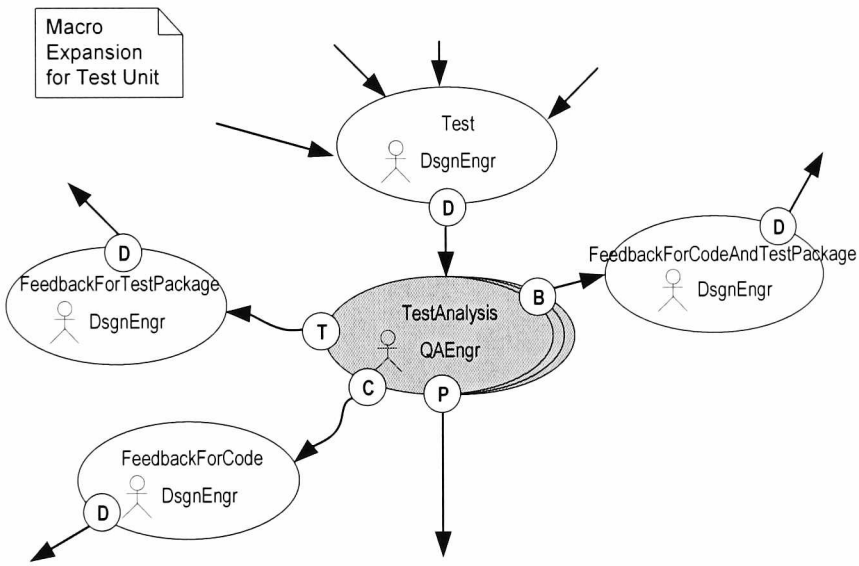


Figure 5-6 Macro Expansion for Test Unit

ISPW-6 Workspaces

Each activity node must always be accompanied by its workspace along with the appropriate definitions of resources and transitions. Additionally, task descriptions can be optionally specified in a workspace using natural language, for example, to

further document the task. Before the workspaces are presented, it should be noted that although task descriptions can be useful to guide the software engineers, they will only be selectively shown here as part of the workspaces defined in the VRPML solution. This is because certain tasks should be relatively straightforward (e.g. Modify Code, Modify Unit Test Package) and should require no further descriptions. Nevertheless, task descriptions will be provided whenever deemed necessary, for example in the case where workspaces host more than one transition and the assigned software engineers are required to make a choice (e.g. Review Meeting, Test Analysis).

In the following paragraph, the workspace definitions will be described for: activity nodes appearing in the main VRPML graph; the sub-graph for the transition Done in Modify Code; and the macro expansion of Test Unit.

The first workspace is for the activity node Modify Design. The workspace, shown in Figure 5-7, consists of a done transition (labelled D), and three artefacts: Current Design with read and write access; Requirement Change with read-only access; and Design Review Feedback with read-only access.

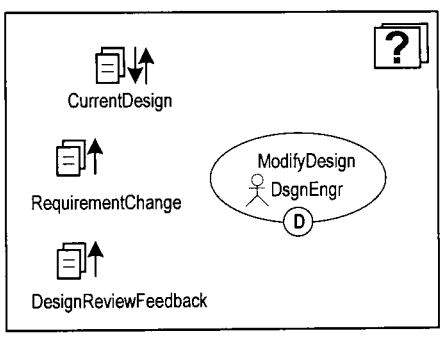


Figure 5-7 Workspace for Modify Design

The next workspace definition is for the multi-instance activity node Review Design. The workspace, shown in Figure 5-8, consists of a done transition (labelled D) as well as a synchronous communication tool to permit inter-person communication between the software engineers involved in review. The two defined artefacts are: Requirement Change with read-only access; and Modified Design with read-only access.

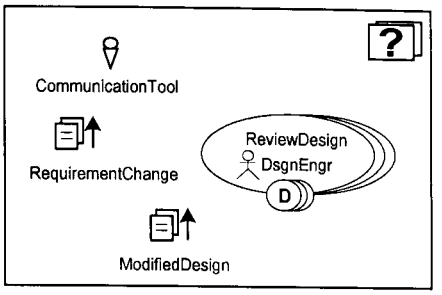


Figure 5-8 Workspace for Review Design

Next, the workspace for a meeting activity node called Review Meeting is defined in Figure 5-9. It consists of two transitions: Passed (labelled P); and Failed (labelled F). Obviously, the two transitions give the two options to the moderator of the Review Meeting as required in the ISPW-6 problem, the description of which can optionally be made available in the task description:

- Transition labelled P should be selected in the case of unconditional approval of the design.
- Transition labelled F should be selected when minor or major changes are recommended. In this case, feedback to the design engineer should be given, and the task Modify Design will be restarted.

Additionally, the workspace also defines four artefacts: Requirement Change with read-only access; Design Review Feedback with read and write access; Outcome Notification with read and write access; and Modified Design with read-only access. Like the workspace for Review Design, the workspace for Review Meeting has a defined synchronous communication tool, as well as an asynchronous communication tool (e.g. email tool) to enable inter-person communication amongst the assigned engineers.

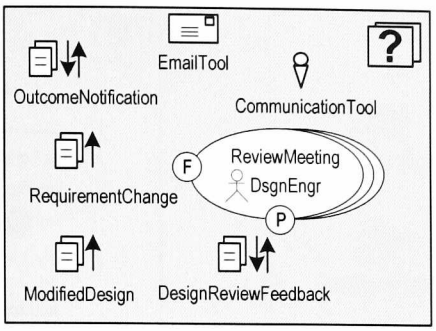


Figure 5-9 Workspace for Review Meeting

The workspaces for different types of activity nodes are uniquely defined not only in terms of the required artefacts but also in terms of their general appearance. This is to inculcate a sense of awareness, that is, process engineers should be able to distinguish at a glance among the different types of activities expressed in a VRPML process model. For example, some activities may concurrently involve other people, such as in the case of meeting and multi-instance activities. In fact, as will be seen in Chapter 6, workspaces for different types of activity nodes are also uniquely represented in their virtual environments during enactment.

The definitions for the rest of the workspace forming the VRPML solution to the ISPW-6 problem are similar and straightforward and are discussed only briefly. The workspace definitions for Modify Code, Modify Test Plans, Modify Unit Test Package and Check Compilation are shown in Figure 5-10.

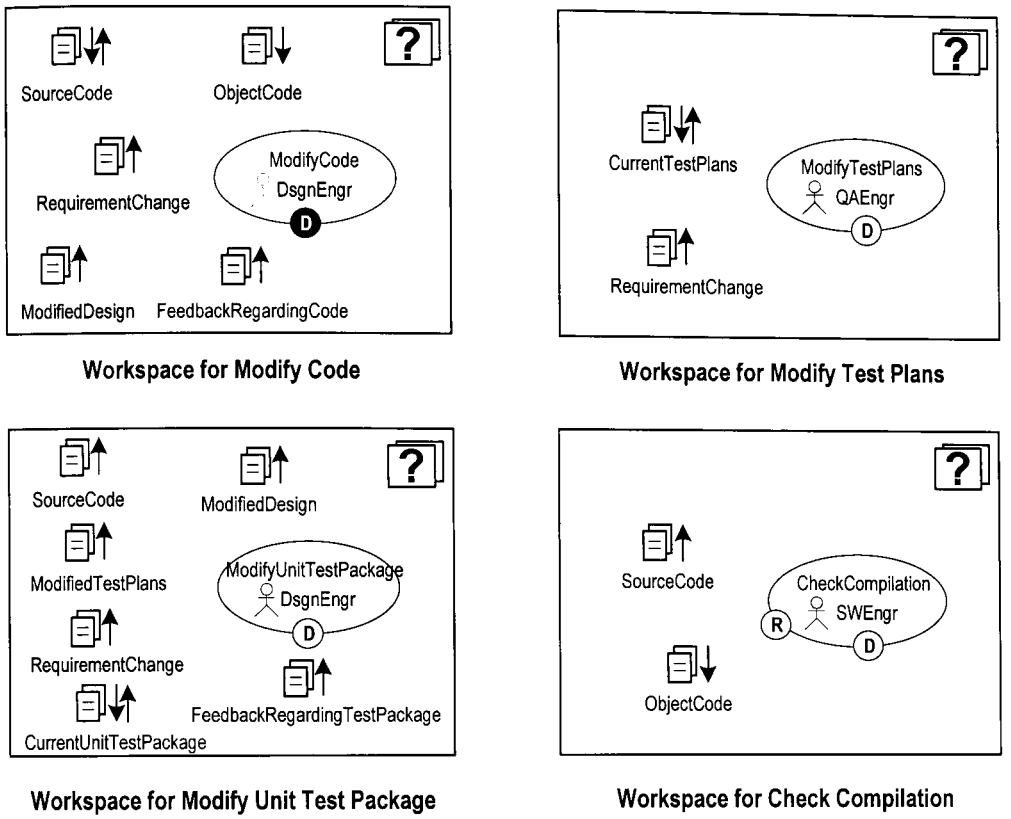


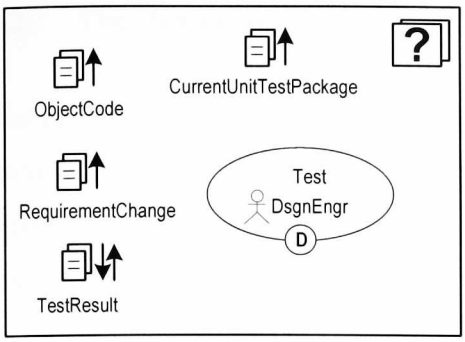
Figure 5-10 Workspaces for Modify Code, Modify Test Plans, Modify Unit Test Package, and Check Compilation

The workspace to note is that of Modify Code as it has a defined decomposable transition Done (labelled D). As discussed earlier, transition D is decomposed into the activity Check Compilation, whose workspace definition is also shown in the same figure.

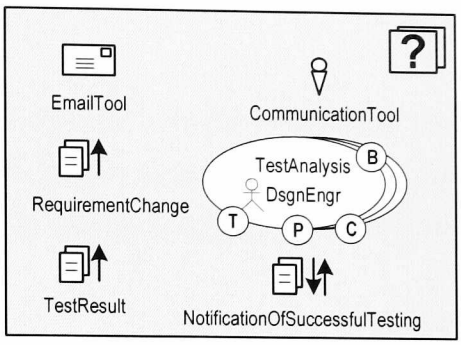
Finally, the workspace definitions for the macro expansion Test Unit are given in Figure 5-11. The workspace to note is that of Test Analysis where four transitions are defined: Passed (labelled P); Code Feedback (labelled C), Test Feedback (labelled T), and Both Feedback (labelled B). The four transitions provide the moderator of Test Analysis with the options as required by the ISPW-6 problem. The description of each transition is given below and can be optionally made available as part of the task description.

- The transition labelled P (Passed) should be selected when the Test Unit is successful, that is, 90% test coverage has been achieved.

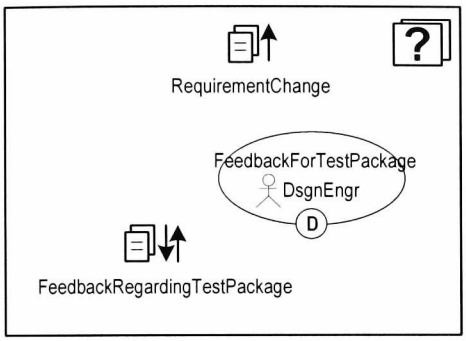
- The transition labelled C (Code Feedback) should be selected when the task Modify Code has to be restarted and feedback regarding code must be furnished.
- The transition labelled T (Test Feedback) should be selected when the task Modify Unit Test Package has to be restarted and feedback regarding the test package must be furnished.
- The transition labelled B (Both Feedback) should be selected when both tasks Modify Code and Modify Unit Test Package have to be restarted and feedback regarding both the code and test package must be furnished.



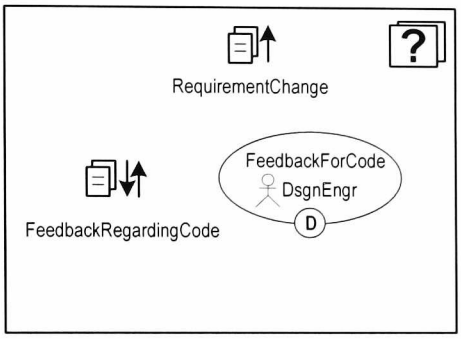
Workspace for Test



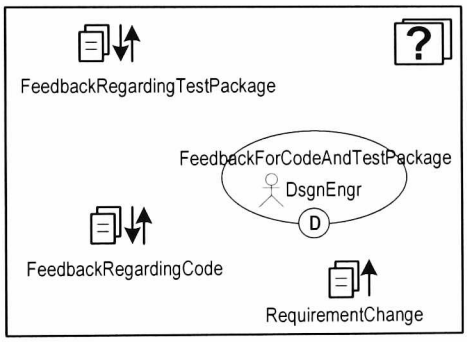
Workspace for Test Analysis



Workspace for Feedback for Test Package



Workspace for Feedback for Code



Workspace for Feedback for Code and Test Package

Figure 5-11 Workspaces for Test, Test Analysis, Feedback for Test Package, Feedback for Code, and Feedback for Code and Test Package

Summing up, this section has presented a step by step solution of the ISPW-6 problem expressed in VRPML. In doing so, issues arising from the VRPML solution were discussed. In the next section, the second case study based on the waterfall development model will be addressed.

5.3 The Waterfall Development Model

The earliest form of the software processes based on the waterfall model [58] were introduced by Royce in 1970 [60]. Since then, many variants from the original waterfall model have been proposed. One of its variants, adopted from DeBellis and Haapala [25], is shown in Figure 5-12.

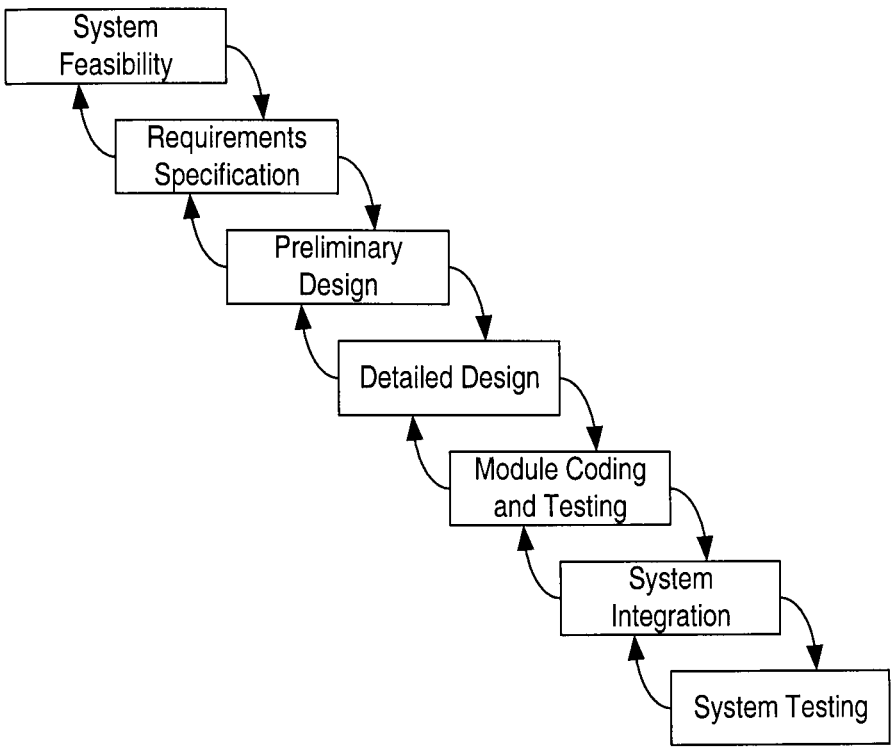


Figure 5-12 The Waterfall Model

Among the characteristics of the waterfall model are:

- The model is divided into a number of separate stages from system feasibility to maintenance.
- Each stage has a clearly delineated activity which is performed in a linear and sequential manner.
- Each stage is also independent that is, there is no overlap amongst stages.
- Feedback is usually provided to the preceding stage.
- The completion of a stage is determined by a review either formally or informally and conducted at the end of each stage so that development can

proceed to the next stage. This is important because the output of the current stage often becomes the input of the next stage.

In order to support modelling and enacting of software processes implementing the waterfall model, each stage of the model must be precisely defined in detail. Building on the work by Sommerville [60] and the waterfall model given earlier, Table 5-2 in the next page summarises the possible activities along with their inputs and outputs. It must be stressed that this is only one of the possible list of activities as there are a number of variations to the waterfall model.

Referring to Table 5-2, the summary of activities involved in each stage of the waterfall model raises a number of issues. Firstly, the roles associated with each defined activity have not been identified. In this case, it is assumed that all of the software engineers involved have the required skills to perform the activities assigned to them. Hence, the role for each activity will be simply software engineers.

Secondly, although the ordering of activities in each stage has not been defined, they can be indirectly inferred from their input dependencies. In fact, activities in a stage can also be enacted in parallel when they are independent, that is, they do not require any input from each other. This will be reflected in the VRPML solution given below.

Finally, in order to highlight only the key aspects of VRPML, only a partial solution of the waterfall model from Table 5-2 will be presented here. The complete solution is provided in the appendix.

Waterfall Stages	Activities	Inputs	Outputs
System Feasibility	Analyse and Define Requirements	Customer Requirements	Draft Feasibility Study Draft Requirement Documents
	Review System Feasibility	Draft Feasibility Study Draft Requirement Documents	Feasibility Study Requirement Documents
Requirements Specification	Prepare Functional Specification	Requirement Documents	Draft Functional Specification
	Prepare Acceptance Test Plan	Requirement Documents	Draft Acceptance Test Plan
	Prepare Draft User Manual	Draft Functional Specification	Draft Preliminary User Manual
	Review Specification	Draft Functional Specification Draft Acceptance Test Plan Draft Preliminary User Manual	Functional Specification Acceptance Test Plan Preliminary User Manual
Preliminary Design	Prepare Architectural Specification	Requirement Documents Functional Specification	Draft Architectural Specification
	Prepare System Test Plan	Requirement Documents Functional Specification	Draft System Test Plan
	Review Preliminary Design	Draft Architectural Specification Draft System Test Plan	Architectural Specification System Test Plan
Detailed Design	Prepare Interface Specification	Functional Specification Architectural Specification Requirement Documents	Draft Interface Specification
	Prepare Integration Test Plan	Functional Specification Architectural Specification Requirement Documents	Draft Integration Test Plan
	Prepare Design Specification	Functional Specification Architectural Specification Draft Interface Specification Requirement Documents	Draft Design Specification
	Prepare Unit Test Plan	Functional Specification Architectural Specification Draft Interface Specification Requirement Documents	Draft Unit Test Plan
	Review Detailed Design	Draft Interface Specification Draft Design Specification Draft Integration Test Plan Draft Unit Test Plan	Interface Specification Design Specification Integration Test Plan Unit Test Plan
Module Coding and Testing	Perform Coding	Requirement Documents Design Specification	Draft Program Code
	Perform Unit and Module Testing	Unit Test Plan Draft Program Code	Draft Unit Test Report
	Review Coding and Testing	Draft Program Code Draft Unit Test Report	Program Code Unit Test Report
System Integration	Perform Integration Testing	Integration Test Plan Program Code	Draft Integration Test Report
	Prepare Final User Manual	Preliminary User Manual Functional Specification Program Code	Draft User Manual
	Review Integration Testing	Draft Integration Test Report Draft User Manual	Integration Test Report User Manual
System Testing	Perform System Testing	System Test Plan Acceptance Test Plan Program Code	Draft System Test Report
	Review System	Program Code User Manual Draft System Test Report	Final Release

Table 5-2 Waterfall Model Activities, Inputs, and Outputs

5.3.1 VRPML Solution of the Waterfall Development Model

The main graph of the VRPML solution for the software processes based on the waterfall model is given in Figure 5-13.

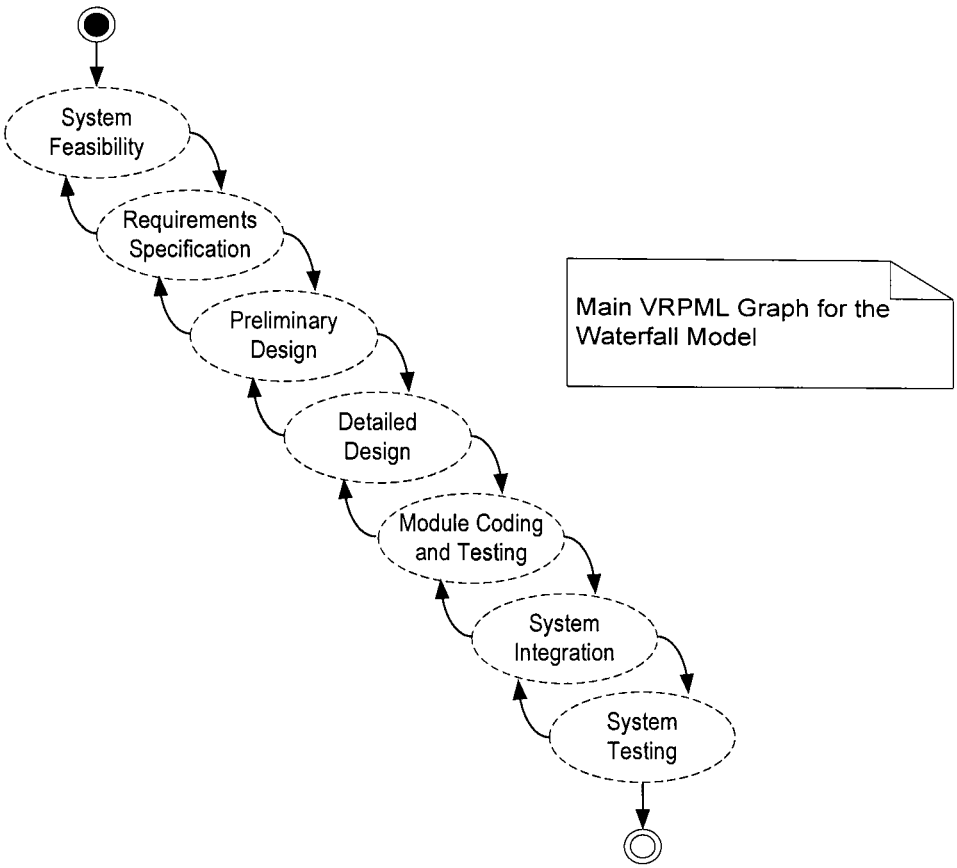


Figure 5-13 Main VRPML Graph for the Waterfall Model

Here, the waterfall model is constructed using the top-down development approach, that is, the modelling starts from the high-level view of macro nodes and works down to activity nodes. This approach is contrary to the approach taken to construct the solution to the ISPW-6 problem in the previous section, where the bottom-up development approach was adopted.

Intentionally arranged like the waterfall model, the VRPML graph in Figure 5-13 clearly resembles the model given in Figure 5-12. The graph consists of seven macro nodes. To further illustrate the VRPML notation, one of the macros, called Detailed Design, is shown in Figure 5-14. The presence of macros related to other stages in the figure is merely to give focus and context to the expansion.

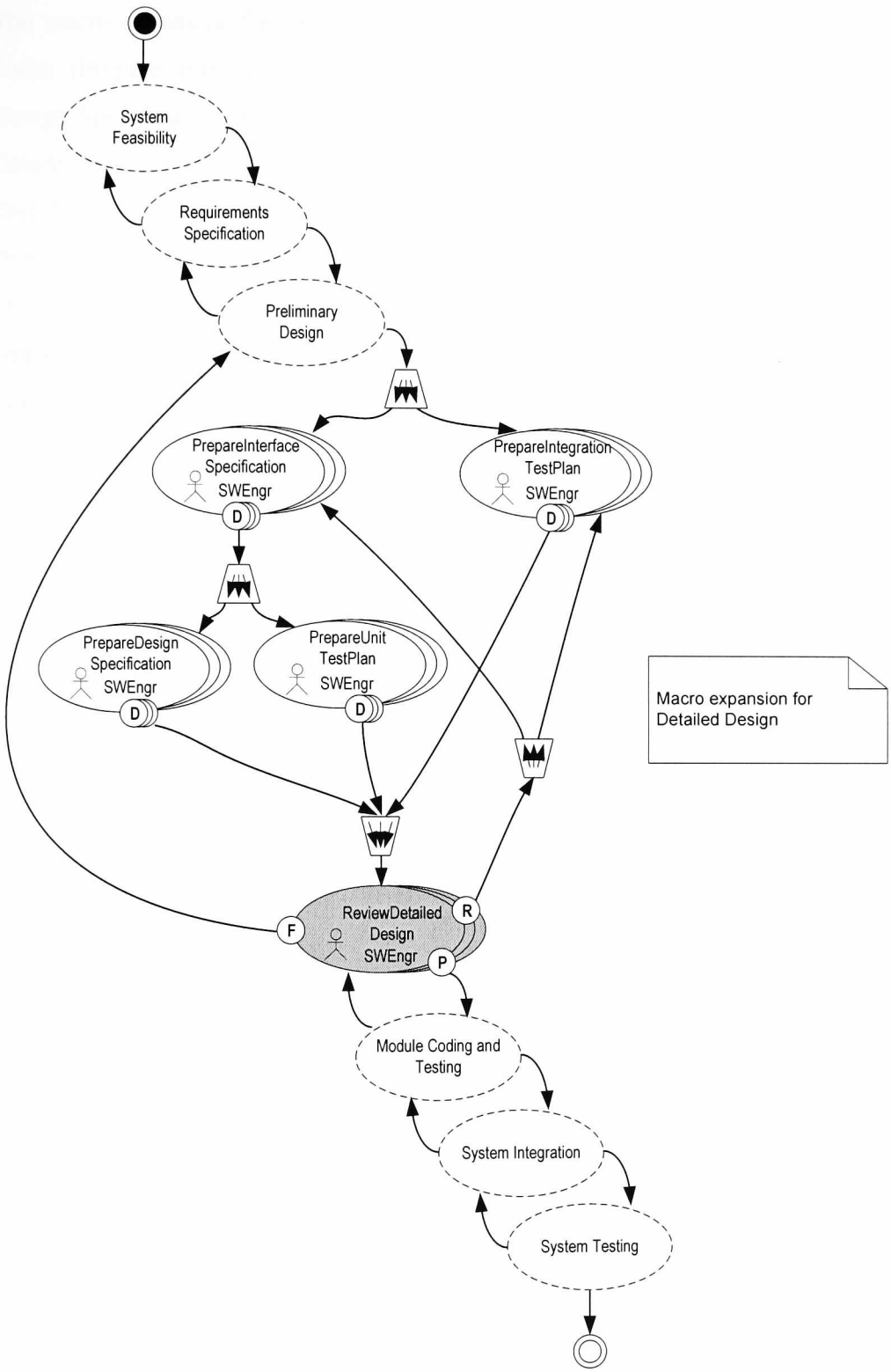


Figure 5-14 Macro Expansion for Detailed Design

The macro expansion for Detailed Design consists of four multi-instance activity nodes (Prepare Interface Specification, Prepare Integration Test Plan, Prepare Design Specification, and Prepare Unit Test Plan) and one meeting node (Review Detailed Design). Because Prepare Interface Specification and Prepare Integration Test Plan are independent of each other, they can be enacted in parallel. However, Prepare Design Specification and Prepare Unit Test Plan can only be enacted after Prepare Interface Specification has been completed. This is because both Prepare Design Specification and Prepare Unit Test Plan require an artefact from Prepare Interface Specification, called the Interface Specification, as one of their inputs (see Table 5-2). Actually, once Prepare Interface Specification has been completed, both Prepare Design Specification and Prepare Unit Test Plan can be enacted in parallel. Lastly, Review Detailed Design is enacted when all the above activities have been completed.

In terms of transitions, Prepare Interface Specification, Prepare Integration Test Plan, Prepare Design Specification and Prepare Unit Test Plan each have only one defined transition for Done (labelled D) to allow their completion. However, Review Detailed Design has three defined transitions: Redo (labelled R) in order to allow loopback to the previous activities; Passed (labelled P) in order to move to the next stage; and Feedback (labelled F) in order to permit feedback to the previous stage.

As far as workspaces are concerned, they can be straightforwardly defined by analysing the inputs for each activity. Therefore, they will not be discussed here but are given in the appendix.

All of the macros given in Figure 5-13 expand into a combination of multi-instance activity nodes and meeting activity nodes connected by arcs. The reason for using multi-instance activity nodes and meeting activity nodes is to demonstrate that VRPML supports the dynamic creation of tasks, that is, no prior assumption is made when constructing the model in terms of how many engineers have to be assigned to any of the activities represented by these nodes.

5.4 Summary

This chapter has shown two well-known software processes problems expressed in VRPML. Importantly, it has demonstrated how the VRPML syntax works to support modelling and enacting of the ISPW-6 problem as well as the waterfall software development model. In Chapter 7, these two VRPML solutions will be revisited in order to evaluate the applicability and usefulness of VRPML. As a complement to the material presented in this chapter, Chapter 6 investigates issues related to implementation and enactment.

Chapter 6 – Implementation Issues

The previous chapter has shown how the VRPML notation supports the modelling of software processes. In particular, process models for the the ISPW-6 problem and the waterfall model have been demonstrated in VRPML.

This chapter focuses on implementation issues in order to investigate whether process models expressed in VRPML can actually be enacted, and whether enactment identifies further issues for consideration. This chapter begins by outlining the main aim and objectives which act as guidelines for performing the implementation. Then, the chapter discusses the runtime support environment which VRPML requires to achieve its enactment. Next, the chapter shows enactment of the ISPW-6 model in order to demonstrate the main features of VRPML. Finally, this chapter closes by giving a short summary.

6.1 Aim and Objectives

The main aim of this chapter is to demonstrate that faithful enactment of the VRPML semantics can be achieved. Considering this aim, the objectives of the chapter are:

- To demonstrate that enactment of the process model expressed in VRPML can be achieved in a distributed environment
- To demonstrate that dynamic creation of tasks and allocation of resources can be supported by exploiting the enactment model
- To demonstrate that integration with a virtual environment is possible at the PML enactment level. Thus, awareness and visualisation issues can be supported.

6.2 VRPML Support Environment

In order to implement VRPML, a number of components for the support environment can be identified. These components and their interactions are shown in Figure 6-1.

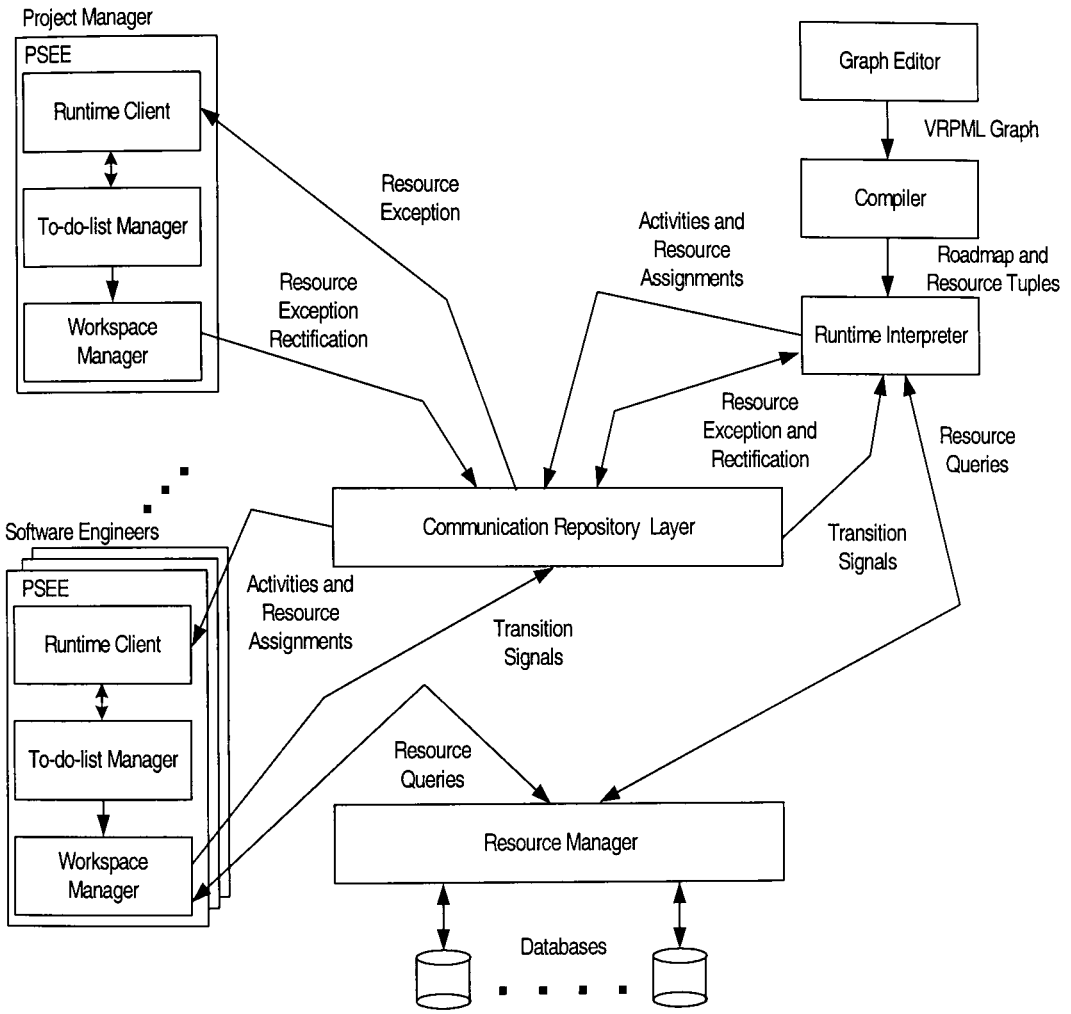


Figure 6-1 VRPML Support Environment

The main components of the complete VRPML support environment are:

- Graph Editor – allows the VRPML graphs to be specified.
- Compiler – compiles the VRPML graphs into some immediate format for enactment.
- Runtime Interpreter – interprets the compiled VRPML graph.

- Runtime Client – retrieve activities and resource assignments from the communication repository layer.
- To-do-list Manager – manages the activities assigned to a particular software engineer.
- Workspace Manager – manages activity workspace in a virtual environment, manages activity transition, and forward queries to the resource manager.
- Communication Repository Layer – allows communication between the runtime interpreter, runtime client, and workspace manager.
- Resource Manager – queries the databases for artefacts.

A complete VRPML support environment would require development of all of the components, a task beyond the time available for the author. Therefore, implementation efforts were concentrated on essential enactment components, as will be discussed below.

Graph Editor

Much like a programmer's editor in a textual programming language, the graph editor would allow VRPML graphs to be drawn and changed. It would also allow browsing through graphs, and would support examination of every level of the graph (for instance, by opening further graph-editor windows onto workspaces and macros) in order to assist awareness issues (i.e. in terms of the readability of the VRPML graph). Although having a dedicated graph editor for VRPML would be vital for a complete system, it warrants no further discussion because the technology of graph editors is essentially already well-established. Consequently, the graph editor for VRPML has not been developed, and VRPML graphs were drawn manually.

Compiler

A compiler would perform syntax checking and translate a VRPML graph into an intermediate format known to the runtime interpreter. In this research work, the compiler for VRPML has not been developed and the compilation of the VRPML graphs was performed manually. However, in order to ensure that a compiler could

be written, research into the information needed for enactment was felt to be necessary. In particular, an important consideration for compiling VRPML is the information stored in the intermediate format. Clearly, the topology of the VRPML graph in terms of the ordering and sequencing of activities together with their resource assignments (if any) needs to be preserved. One solution shown in Figure 6-2 is to produce a *roadmap* of how activities are interconnected and to generate the resource assignments in all the workspaces separately as *resource tuples*.

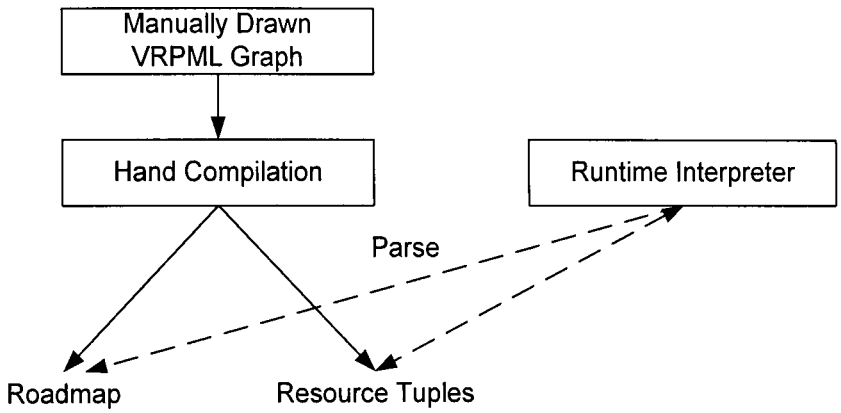


Figure 6-2 Compilation of the VRPML Graph

A format for the roadmap and the resource tuples has been identified (described below) and used to facilitate the hand-compilation of the VRPML graph, and hence permit enactment. To illustrate this technique, Figure 6-3 shows an example graph to be compiled.

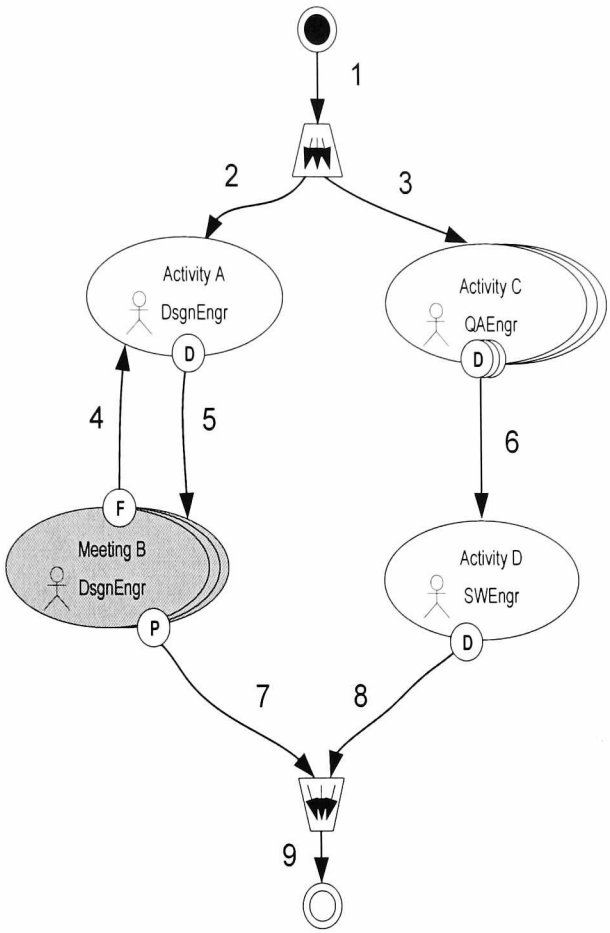


Figure 6-3 Simple VRPML Graph

The roadmap that is generated for the above VRPML graph is as follows (where the number shown on each arc is the internally generated control-flow signal id which is allowed to flow through the arc):

```
1,       Master 2]) Master] 3])
2,       Administrator] Activity A])
3,       Administrator] Activity C])
4,       Administrator] Activity A])
5,       Administrator] Meeting B])
6,       Administrator] Activity D])
7, 8,    Master] 9])
9,       Master] Terminate])
```

A number of items in the roadmap need clarification and several terms in the roadmap need to be defined. "Master" refers to the runtime interpreter itself whilst "Administrator" refers to the role in charge of rectifying resource exceptions (e.g. when resource assignments are not specified or not available). The characters , ,]

and) merely serve as separators which are used by the runtime interpreter to parse the enabling control-flow signal id (shown as a unique number for clarity), the defined activities and the target activity assignment (e.g. master or administrator).

Resource tuples must be generated for each activity defined in the graph. To illustrate the contents of a resource tuple, assume that the workspace for activity A shown in Figure 6-3 is defined below:

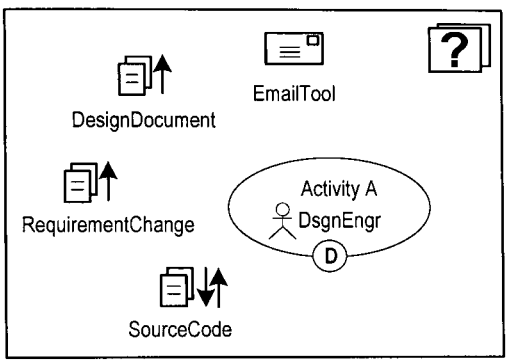


Figure 6-4 Example Workspace

The resource tuple for activity A is generated as follows with keywords (shown in bold) used for clarity and human readability.

ActivityName = Activity A, 2,
ActivityType = General Purpose,
Role = DsgnEngr
AssignedEngineer = Unspecified,
Artefact = Design Document, Path/Url for Modified Design, Read, Path/Url for tool,
Artefact = Requirement Change, Path/Url for Req. Change, Read, Path/Url for tool,
Artefact = Source Code, Path/Url for Source Code, Read/Write, Path/Url for tool,
Tool = Email Program, Email, Path/Url for tool,
Transition = D, Transition Done, Non-Decomposable, 5,
Description = Put the description of the activity here.

A resource tuple carries runtime information about the workspace consisting of: activity name and type; resource assignments including access rights for artefacts; tool assignments; and the defined transitions as well as the id of each control flow that will be generated if a particular transition is selected. One important aspect to observe in order to generate the resource tuple is that the id of each control-flow

signal to be generated must be consistent with that defined in the roadmap. For example, transition Done must generate the control-signal id 5 in order to enable activity B.

Using the roadmap and resource tuples, enactment can easily be achieved. In fact, by adopting the Linda tuple space [33] as the communication repository layer, enactment can be achieved in a distributed environment. This issue will be further developed when the experimental setup is discussed.

Runtime Interpreter

Having considered the graph editor and the compiler, the next component is the runtime interpreter. Much of the functionality has already been implied in the earlier discussion. The full list of the runtime interpreter's functions is as follows:

- parse, maintain, and interpret the runtime information held in the roadmap and the resource tuples
- check for the arrival of control-flow signals in the communication repository layer, and decide when activities are able to fire
- interact with the resource manager to check for resource availability before enabling activities and return exceptions accordingly
- detect the termination of enactment and shut down gracefully

Runtime Client

To support enactment in a distributed environment, there is the need to implement a runtime client which works on behalf of the to-do-list manager (described below) in order to retrieve the activities and their resource allocations from the communication repository layer according to a software engineer's assignments. There are two types of runtime client which can be considered. In the first type, the runtime client automatically retrieves activities and their resource allocations as soon as they are assigned. In this case, there is a need for a dedicated channel which maintains a connection between the runtime client and the communication repository layer at all times. In the second type, the runtime client only retrieves activities and their resource allocations when there is an explicit request from the

software engineer. As far as this research work is concerned, both types are equally useful. However, the second type has been chosen because it simplifies the implementation in the sense that there is no need to setup a dedicated communication channel between the runtime client and the communication repository layer.

To-do-list Manager

In order to manage the assigned activities, there is also a need for a to-do-list manager. The full list of the to-do-list manager's functions is as follows:

- create a to-do-list for a particular software engineer
- interact with the runtime client upon request to retrieve the assigned activities from the communication repository layer
- provide an internal to-do-list queue to store the assigned activities received from the runtime client
- manage the graphical user interface (GUI) to allow a software engineer to select, browse, or undertake activities from the to-do-list queue as well as retrieve the assigned activities from the communication repository layer
- forward an activity and its resource allocations to the workspace manager (described below) when the activity is selected to be undertaken
- provide an interface to quit the to-do-list

As an illustration, Figure 6-5 shows a sample snapshot of the to-do-list GUI for a software engineer name Kamal where the current activity in the to-do-list queue is Review Design. Five buttons are also shown: Retrieve the activity; Perform the activity; Quit the to-do-list; Set; and Send. These buttons implement the functions described in the bulleted list above with the exception of the Send button. The Send button will be described below when the discussion on the workspace manager has been developed. It must be stressed that implementation efforts have concentrated on functionality rather than the aesthetics of the interface.

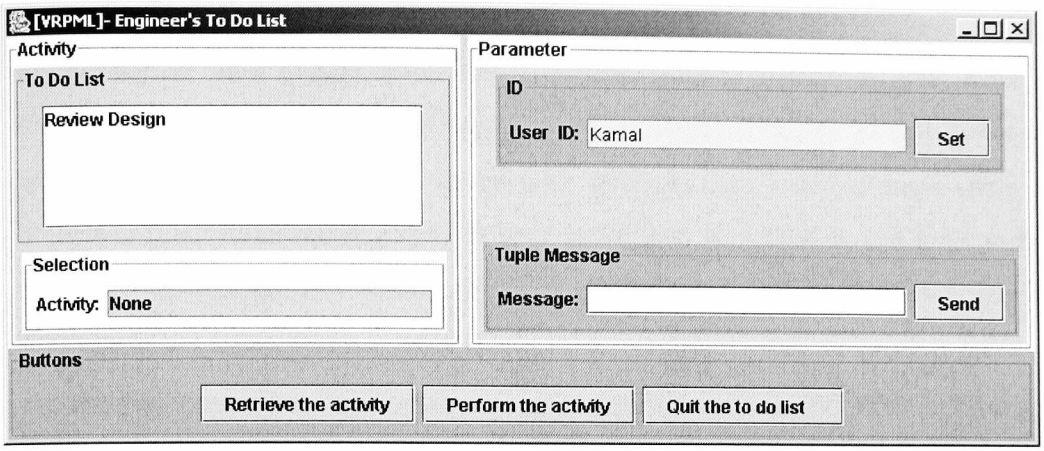


Figure 6-5 The To-do-list Graphical User Interface

Workspace Manager

The workspace manager generates and maintains each activity's workspace according to its work context, that is, in terms of the artefacts and tools required to complete that activity. As discussed in Chapter 2, maintaining work context is important to give the software engineers a sense of awareness about the activity that they are currently undertaking.

At a first glance, generating and maintaining each activity's workspace according to its work context seems like a difficult task. However, a closer look reveals that this can easily be achieved by the workspace manager parsing the runtime information stored in the resource tuple. Using this runtime information, each activity's workspace can be generated when it is enabled.

As far as the implementation is concerned, the chosen approach to generate the workspace for each activity in a virtual environment was through the use of the Virtual Reality Modelling Language (VRML), a language for specifying three dimensional scenes with rich sets of object primitives and events. With VRML, the workspace can be translated to a VRML scene, and the actual translation may be facilitated by a freely available CyberVRML97 library package [49] integrated as part of the workspace manager itself. In this work, efforts have been directed towards providing functionality and ignoring the aesthetics of the virtual environment.

Figure 6-6 below depicts the possible translation of the workspace and representation in a virtual environment for an activity called Modify Design. Utilising Doppke's task-centered mapping (discussed in Chapter 3), the activity's workspace maps into a virtual room with artefacts and tools corresponding to objects in that virtual room. Artefacts are represented as cylinders, and their access rights are distinguished by colours. Task descriptions are represented as help boxes. Transitions are represented as spheres.

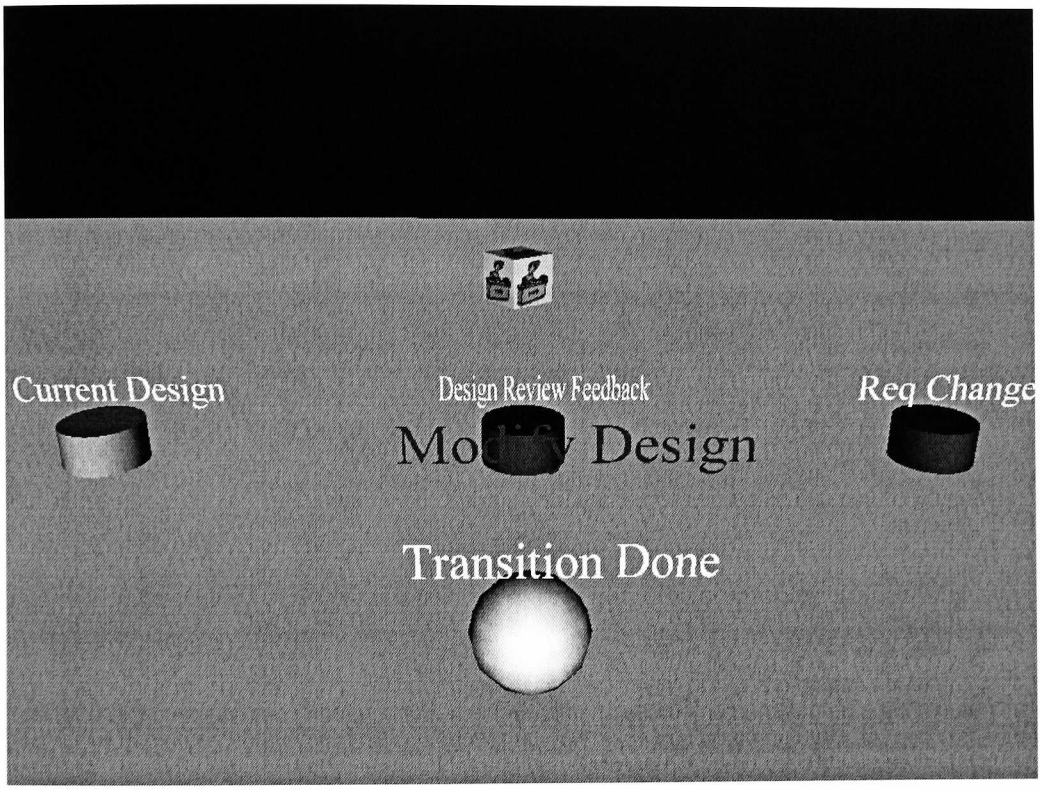


Figure 6-6 Sample Workspace in a Virtual Environment

Following a request from the software engineer, the workspace manager needs to interact with the resource manager in order to query the databases for artefacts and tools in the workspace to perform the activity. Because VRML supports event handling, it would be possible for the software engineer to access the objects (e.g. artefacts, tools, and transitions) in a scene, although this has not been investigated further as the objective here is to demonstrate that it is feasible to support integration with a virtual environment at the PML enactment level.

However, the fact that in the prototype the representation of transitions in the virtual environment are not active means that a mechanism is needed to simulate transitions and hence indicate the completion (or cancellation) of an activity. Thus, a Send button and a message box are provided in the to-do-list GUI (see Figure 6-5) to achieve the sending of a control flow signal to the communication repository layer by the workspace manager.

Communication Repository Layer

The main function of the communication repository layer is to act as an intermediate mailbox for keeping the assigned activities and their resource tuples as well as the control-flow signals. There are three main components which interact with the communication repository layer: the runtime client to allow query of activity assignments and their resource tuples; the runtime interpreter to allow assignment of activities and their resources allocations to be made; and the workspace manager to allow the control-flow signals generated from transitions to be sent.

As far as implementation is concerned, the distributed shared memory model based on the Linda tuple space seems to be a suitable choice for the communication repository layer. The main reason for choosing the Linda tuple space stemmed from the fact that the VRPML enactment model (discussed in Chapter 4) is based on Linda, the base language from which the Linda tuple space is derived. In addition, Linda provides several pre-defined primitives which facilitate pattern matching of tuples in the tuple space and they can be used to simplify the implementation. While there are many Linda implementations available, Jada [17], the Linda implementation based on Java, has been chosen for this research work. Jada permits the user to setup a client-server based Linda tuple space that uses Java Remote Method Invocation. As will be seen later in this chapter, it is this tuple space that facilitates enactment in a distributed environment.

Resource Manager

The last component that needs to be considered for supporting enactment is the resource manager. The main function of the resource manager is to handle the

queries received from the runtime interpreter and the workspace manager. As discussed earlier, the queries received from the interpreter mainly involve checking for resource allocation and availability before allowing that activity to be assigned. The queries received from the workspace manager mainly involve requests to manipulate existing artefacts and tools or to create new artefacts. In addition to handling queries, the resource manager also enforces the required access rights involving artefacts, and supports changes and updates of the shared artefacts by multiple software engineers.

In terms of implementation, the resource manager can be straightforwardly realised by a database server with the capability of accessing more than one database at a time in a distributed environment. Because the technology to access the database server is well-established, it warrants no further discussions.

6.3 Experimental Setup

Having identified the main components of the runtime system, this section focuses on the experimental setup. It must be noted that the prototype VRPML support environment in its current form is not suitable to support the real-world software engineering activities, and its purpose is to gain insights into the actual enactment.

In this experimental setup, the VRPML process model for the ISPW-6 problem was adopted for enactment, although only partial enactment will be discussed here. Referring to Figure 6-7 below, enactment will be demonstrated for the following activities: Modify Design; Modify Test Plans; Review Design; and Review Meeting. Furthermore, only resource allocation via role assignment will be considered.

Another aspect which has also not been addressed in this experiment is the fact that when the resource exception of a particular activity is raised, that activity can also be optionally aborted without having to rectify that exception. Because the mechanism to abort an activity is a straightforward one (i.e. simply by removing it from the administrator's to-do-list queue and manually generating the required control flow signal), it was not considered further.

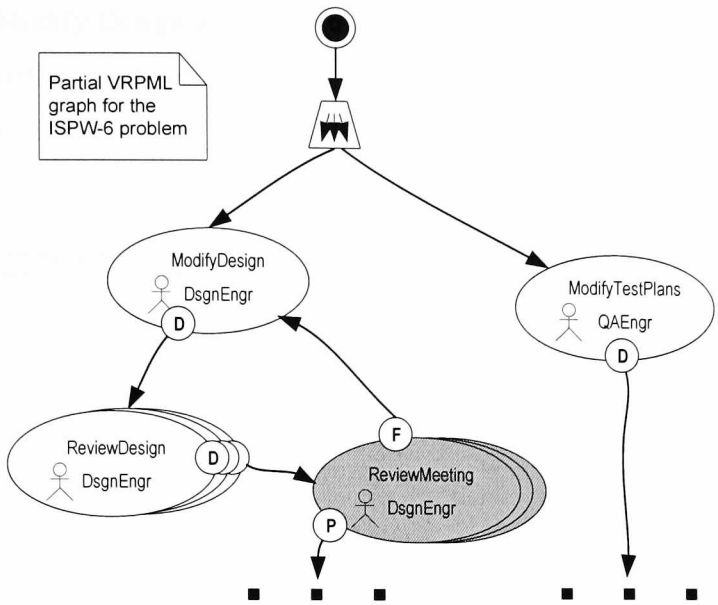


Figure 6-7 Partial Enactment of the ISPW-6 Problem

It is assumed that the above activities involve three software engineers who will be taking on different roles – they are: Kamal, Pete, and Jon. Kamal and Jon will be taking the role of design engineers (abbreviated as DsgnEngr). Pete will be taking the role of quality assurance engineer (abbreviated as QAEngr) and administrator (or process engineer). It is also assumed that Modify Design is pre-assigned to Jon whilst Modify Test Plans, Review Design, and Review Meeting are dynamically assigned. Finally, Kamal, Pete, and Jon are physically isolated, that is, each of them has access to their to-do-list from a separate machine in a distributed environment.

Enactment in this experimental setup is achieved by the runtime interpreter, based on the runtime information in the roadmap and resource tuples generated by compiling the above graph manually. Enactment starts when the start node produces the necessary control-flow signal. In turn, this control-flow signal will cause the replicator node to produce two more control-flow signals. Upon receiving these two control-flow signals, the interpreter queries the resources assignments for Modify Design, and Modify Test Plans in order to put them in the tuple space. Modify Design has already been assigned to Jon, but a resource exception will be thrown for Modify Test Plans. As a result, Modify Test Plans will be automatically assigned to the administrator (i.e. Pete) so that the resource exception can be

rectified. Modify Design and Modify Test Plans will appear on Jon's to-do-list and the administrator's to-do-list respectively as soon as they made the request to retrieve the activity in the tuple space. As an illustration, Figure 6-8 depicts Jon's to-do-list.

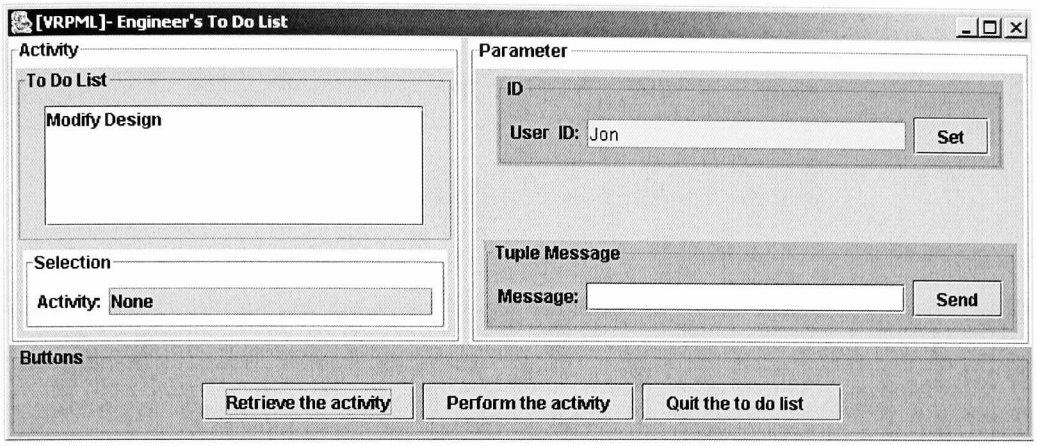


Figure 6-8 Jon's To-do-list

When Jon selects and undertakes Modify Design from his to-do-list, a workspace for Modify Design is automatically opened in a virtual environment. The workspace for Modify Design is similar to that shown in Figure 6-6 and is not shown here. However, the workspace for Modify Test Plans requires further discussion to illustrate how the rectification of resource exception achieves dynamic allocations of resources. When the administrator selects and undertakes Modify Test Plans from his to-do-list, a workspace to rectify the resource exception can be opened in a virtual environment. As shown in Figure 6-9 below, this experimental setup simply uses a text editor to facilitate the updating of resources assignment. Here, the assigned engineer has been allocated to Kamal.

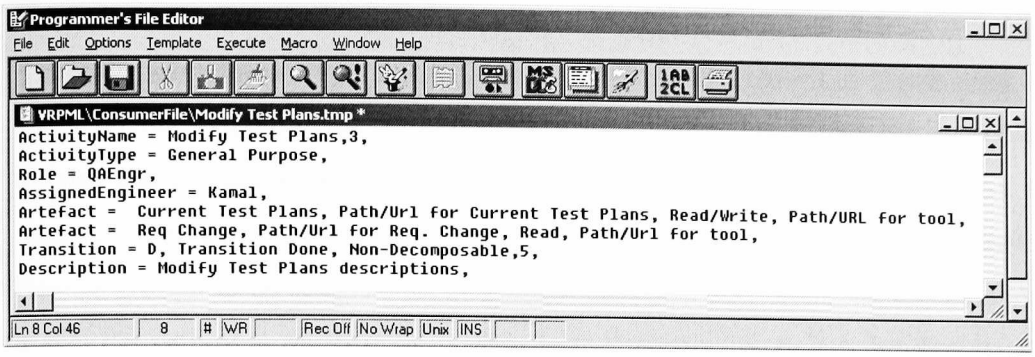


Figure 6-9 Resource Allocations for Modify Test Plans

Once resource allocation has been completed, Modify Test Plans will be put back into the tuple space. This is achieved via the administrator's to-do-list shown in Figure 6-10. Ideally, the administrator's to-do-list would be no different to an ordinary to-do-list, as the resource update should be done automatically in the background by the workspace manager. However, the administrator's to-do-list GUI is tailored to allow the resource update to be put back manually into the tuple space through the update button.

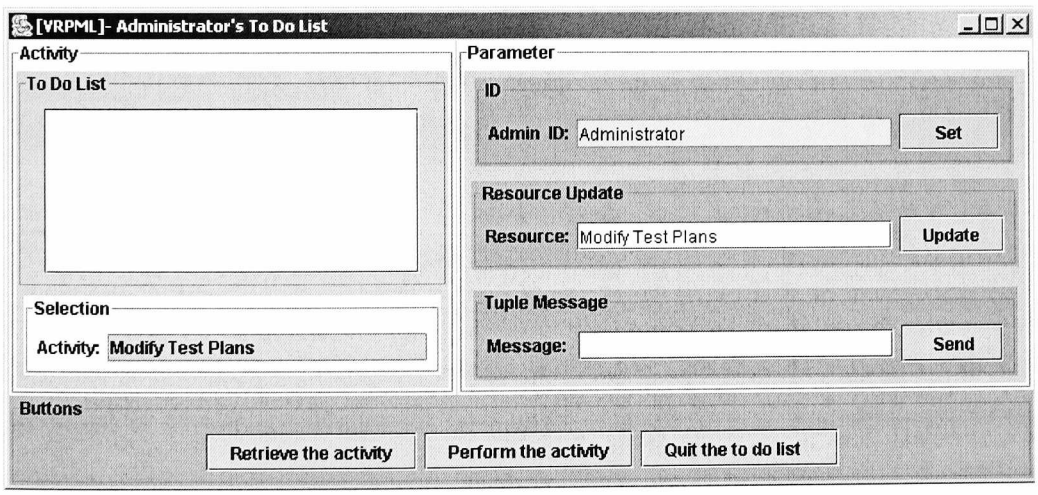


Figure 6-10 Administrator's To-do-list

Once the resource allocation for Modify Test Plans has been updated, Modify Test Plans is now assigned to Kamal. Consequently, when Kamal makes the request to retrieve the activity from the tuple space, Modify Test Plans will appear in Kamal's to-do-list.

Going back to Jon, once he has completed Modify Design and selects the done transition (simulated by the Send button), another control-flow signal will be generated in the tuple space. Upon receiving this control-flow signal, the interpreter queries the resources assignments for Review Design (see Figure 6-7). As no resource assignments have been made, a resource exception will be thrown causing Review Design to be assigned to the administrator. In turn, when the administrator makes the request to retrieve the activity from the tuple space, Review Design will appear in the administrator's to-do-list. Similar to the case of Modify Test Plans discussed earlier, in order to rectify this resource exception there is a need to update the resource tuple for Review Design. As Review Design is a multi-instance activity node, it can be assigned to more than one software engineer through changing its depth, as illustrated in Figure 6-11.

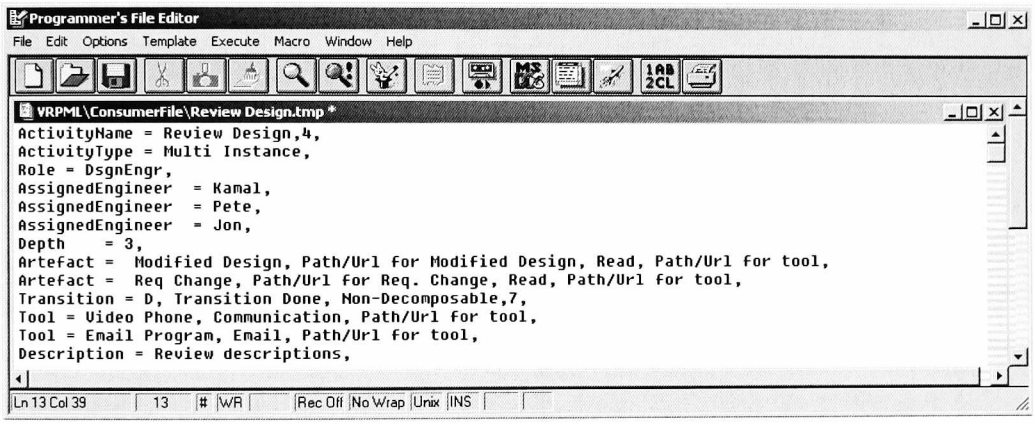


Figure 6-11 Resource Allocations for Review Design

By manipulating the Review Design's depth, the number of software engineers required to review the design (or how many Review Design activities are created) can be tailored according to the current needs of the project. In this example, Review Design is assigned to three software engineers: Pete, Kamal, and Jon. Once resource allocation for Review Design has been completed, it will be put back into the tuple space through the administrator's to-do-list. As a result, Review Design will appear in the to-do-list for Pete, Kamal, and Jon when they make the requests to retrieve the activity from the tuple space.

In order to inculcate the sense of process awareness, the virtual environment representing workspaces for a multi-instance activity node such as Review Design has a different appearance to a workspace for a general purpose activity node (see Figure 6-6) in terms of the background of the workspace.

As far as the completion of Review Design is concerned, because it is a multi-instance activity node assigned to Pete, Kamal, and Jon, all of them must complete the review by selecting the Done transition in their own separate workspaces. Only after all of the done transitions have been selected can a new control-flow signal be generated in the tuple space to enable the subsequent activity Review Meeting.

Finally, upon receiving the control flow signal generated above, the interpreter queries the resources assignments for Review Meeting (see Figure 6-7). As no resource allocations have been made, a resource exception will be thrown causing Review Meeting to be assigned to the administrator. Consequently, when the administrator makes the request to retrieve the activity from the tuple space, Review Meeting will appear in his to-do-list.

Being a meeting activity node, Review Meeting also has an associated depth which can be manipulated in order to allocate engineers dynamically based on the needs of the activity. Using the allocation mechanism described above, the administrator can assign Review Meeting to Pete and Jon, with Pete being the moderator.

As far as the workspace is concerned, being a meeting activity node, Review Meeting can have a different appearance as compared to other types of activity nodes, again, to inculcate the sense of process awareness. In terms of the transitions associated with Review Meeting, these are only accessible to Pete as he is the moderator. Therefore, it is Pete who has the final say of whether the modified design is endorsed or rejected, and only one control-flow signal will be generated as a result.

If Pete decides to choose the Failed transition, the interpreter reassigns Modify Design to Jon. Assuming Jon is still part of the development team, Modify Design will appear in Jon's to-do-list after he makes the request to retrieve the activity from the tuple space. Otherwise, the resource exception will be thrown. After Jon completes Modify Design, Review Design will now be reassigned to Pete, Kamal,

and Jon. After Pete, Jon, and Kamal complete Review Design, Review Meeting will be reassigned to Pete and Jon. This "looping" sequence of activities will continue until Pete, being the moderator, selects the Passed transition after completing Review Meeting.

6.4 Summary

A number of implementation issues have been discussed in this chapter. The main components of the VRPML support environment have been identified and a working prototype has been developed. Using this prototype, an experiment has been conducted which demonstrated enactment by utilising the partial ISPW-6 problem.

Overall, the experiment has successfully achieved its objective of demonstrating enactment in a distributed environment of a process model expressed in VRPML. Furthermore, the experiment has also demonstrated the VRPML support for the dynamic allocation of resources as well as highlighted the possible support for visualisation and awareness issues. It is believed that this experiment has demonstrated that VRPML can be used in practice to express a process model and support its enactment.

While the conversion from the VRPML graph to an intermediate format known to the interpreter was done manually in the experiment, the translation process was done as a compiler would have. No expert knowledge, that is making use of information not available to the compiler, was applied during the translation process. Therefore, it is believed that an automated compiler support for VRPML is implementable. Furthermore, as no major hurdles are encountered when developing the prototype support environments, a complete VRPML support environment appears to be practicable.

Building on results and experiences in this chapter, the next chapter evaluates VRPML in detail in order to highlight its strengths and limitations.

Chapter 7 – An Assessment of VRPML

The previous chapter has demonstrated that VRPML syntax and semantics behave as expected, and enactment can be achieved in a distributed environment. Additionally, the previous chapter has also demonstrated the VRPML support for the dynamic allocation of resources (i.e. by exploiting the enactment model) and the feasibility of supporting awareness and visualisation issues at the VRPML enactment level (i.e. integrating with a virtual environment).

Based on the experience gained from the work described in the previous chapter as well as the modelling of the case study problems discussed in Chapter 5, this chapter presents an assessment of VRPML. Comparison and contrasts between VRPML's features and those of other PMLs will also be made, using the ISPW-6 problem as the benchmark where possible.

7.1 Lessons Learned About VRPML

The main issues which are under consideration in this section relate to the applicability and suitability of VRPML for modelling and enacting software processes. Such consideration may help improve VRPML by providing the necessary feedback based on the author's practical experiences [71].

As far as the assessment is concerned, VRPML will be evaluated on three aspects: the language syntax and semantics, the choice of computational model, and the novel language claims. This assessment is the subject of the next sub-sections.

7.1.1 Language Syntax and Semantics

The main focus of this section relates to the effectiveness of representation and the expressiveness of the VRPML notation. Where appropriate, suggestions are made for VRPML improvements.

Effectiveness of representation

It is believed that the VRPML solutions to the ISPW-6 problem and the software processes based on the waterfall model have demonstrated that VRPML notation is simple and intuitive. Among the reasons are that VRPML adopts a visual syntax, and the general structure of the VRPML graphs resembles the structure of a conventional flowchart.

Although the UML activity diagram [59] is non-enactable, it can be compared to VRPML in terms of its graph representation. Figure 7-1 depicts an example of a software process expressed using the UML activity diagram.

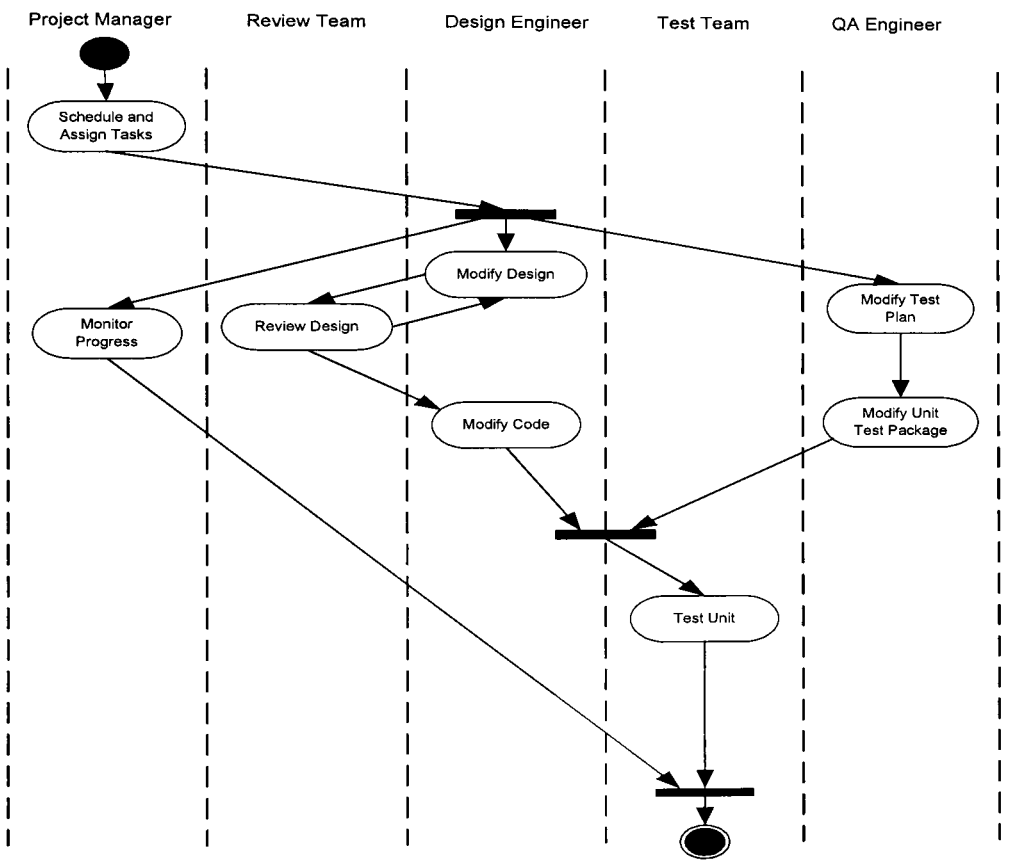


Figure 7-1 Example of the UML Activity Diagram

The UML activity diagram representation of a software process is simple and intuitive. Nonetheless, while the UML activity diagram can be used to express activities in a software process, it lacks features to express the individual role,

resources, workcontexts, and the completion of activities. Thus, the UML activity diagram is not particularly suitable as a PML.

Going back to Table 2-2 in Chapter 2, several of the PMLs surveyed in this thesis also employ a visual notation. They include: FUNSOFT nets [29]; SLANG [9]; Dynamic Task Nets [37]; Little JIL [66]; EVPL [36]; APEL [24]; and PROMENADE [57]. Since Little JIL is the only visual PML for which a detailed solution for the ISPW-6 problem is available for consideration, comparisons will be mainly focused on Little JIL.

Figure 7-2 reproduces part of the Little JIL solution to the ISPW-6 problem [50] involving the activity Modify and Review Design.

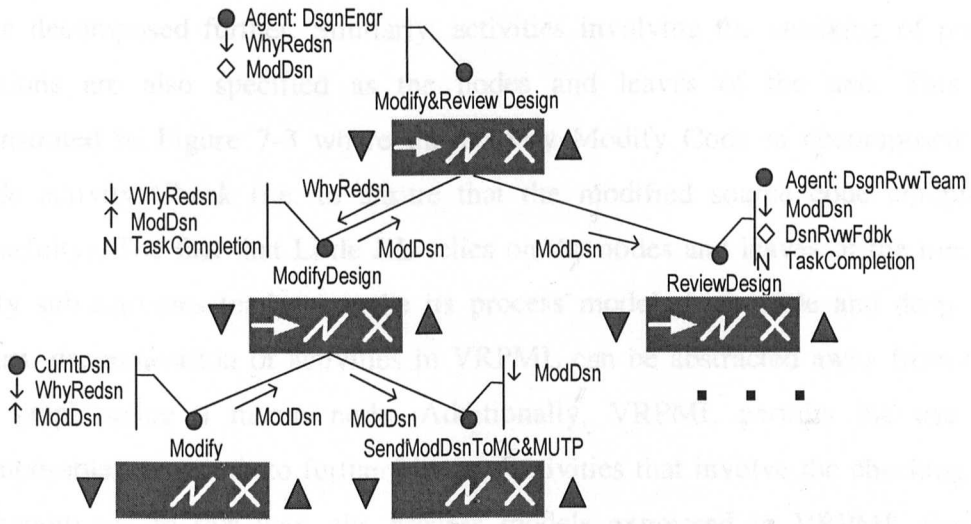


Figure 7-2 Excerpt of the Little JIL Solution to the ISPW-6 Problem

Referring to Figure 7-2, it can be seen that the overall Little JIL notation is quite simple and intuitive. Nevertheless, unlike VRPML, Little JIL does not make any visual distinction between activity types. For example, Review Design in Figure 7-2 uses the same notation as other activities although it represents a decision making activity involving more than one software engineer. Therefore, it can be difficult at a glance to accurately interpret a process model expressed by Little JIL. In VRPML, there are three visibly distinct types of activity (i.e. general purpose activity nodes, meeting nodes, and multi-instance activity nodes). The effect of

distinguishing different types of activities is believed to improve the readability of the VRPML graph.

Being visual languages, both VRPML and Little JIL are likely to suffer from the problem of scale, that is, the process models may take a lot of space. However, the problem of scale appears to be more significant in Little JIL than in VRPML. One reason is due to the way that Little JIL supports the decomposition of activities into sub-activities. As activities are represented in a tree structure, activities in Little JIL are represented as the nodes of the tree with sub-nodes representing decomposition, and leaves at the bottom representing the actual activities assigned to software engineers. For example, in Figure 7-2, the activity Modify & Review Design is decomposed into the activities Modify Design and Review Design which in turn can be decomposed further. Similarly, activities involving the checking of post-conditions are also specified as the nodes and leaves of the tree. This is demonstrated in Figure 7-3 where the activity Modify Code is decomposed to include activity Check (i.e. to ensure that the modified source code compiles successfully). The fact that Little JIL relies on the nodes and leaves of the tree to specify sub-activities tends to make its process models both wide and deep. In contrast, decomposition of activities in VRPML can be abstracted away from the main graph using a macro node. Additionally, VRPML permits the use of decomposable transitions to further abstract activities that involve the checking of post-conditions. In this way, the process models expressed in VRPML should occupy less space than a Little JIL equivalent.

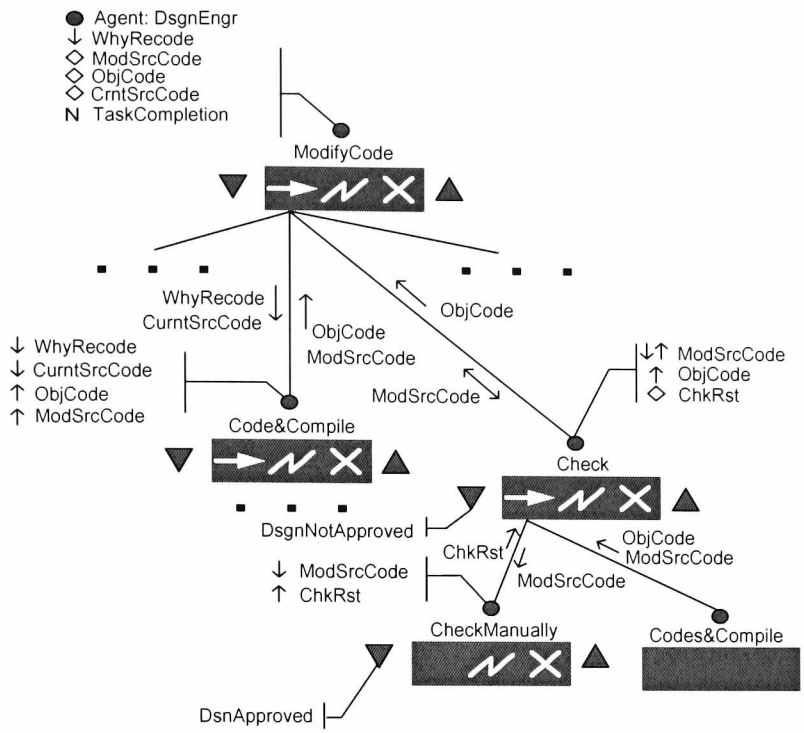


Figure 7-3 Excerpt of the activity Modify Code in Little JIL

Another important difference between VRPML and Little JIL relates to a separation of concerns. Unlike VRPML, Little JIL depicts both the control-flow and the data-flow in the same view. Although depicting both flows might be useful to give the overall context about a particular activity, a counter argument suggests that it results in a cluttered view. In VRPML, control-flow and data-flow are completely separated into the graphs and workspaces respectively, essentially giving two different views: the flow of activities and the individual activity's work context. The flow-of-activities view gives the overall dependencies amongst different activities whilst the activity's work context view gives focus to an activity in question. These views are advantageous to assist reasoning about a software process as well as to support awareness and visualisation issues.

Finally, as noted by Lee [50], Little JIL is not always suitable for top-down modelling because in certain situations process abstraction and control abstraction in the Little JIL step can be in conflict with each other, causing the need to restructure the overall process model when that conflict occurs. VRPML, on the

other hand, supports both top-down and bottom-up modelling without any problems.

Despite the above advantages, it is clear that there are a number of aspects of VRPML that could be improved. In applying the macro nodes to the modelling and enacting of the case study problems, it was found that while a macro can improve the readability of the VRPML graph in question by grouping nodes together, it can be a somewhat difficult notation to employ especially in the context of achieving reuse. The difficulty of a macro node stems from the fact that there is a need to trace out its expansion in terms of arcs going in and coming out, in order to ensure that proper connections of nodes can be made every time that macro is used. This is demonstrated by the macro Test Unit from the ISPW-6 problem in Figure 7-4.

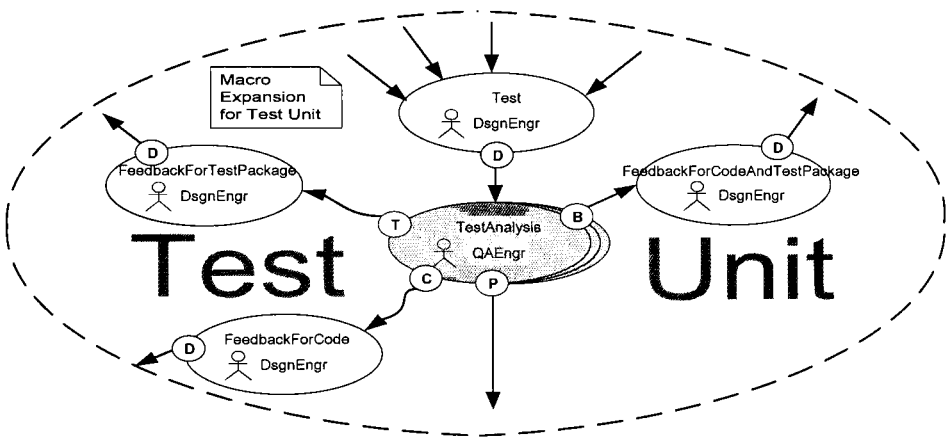


Figure 7-4 Macro Test Unit Revisited

To address this difficulty, labelled connection points could be introduced as part of the macro for designating input and output arcs so that a macro can be treated as a black box reusable component. As an analogy, labelled connection points can be thought of as passing parameters to a procedure in a textual programming language. This potential amendment to a macro node is demonstrated in Figure 7-5. Here, white circles indicate input connection points and grey circles indicate output connection points.

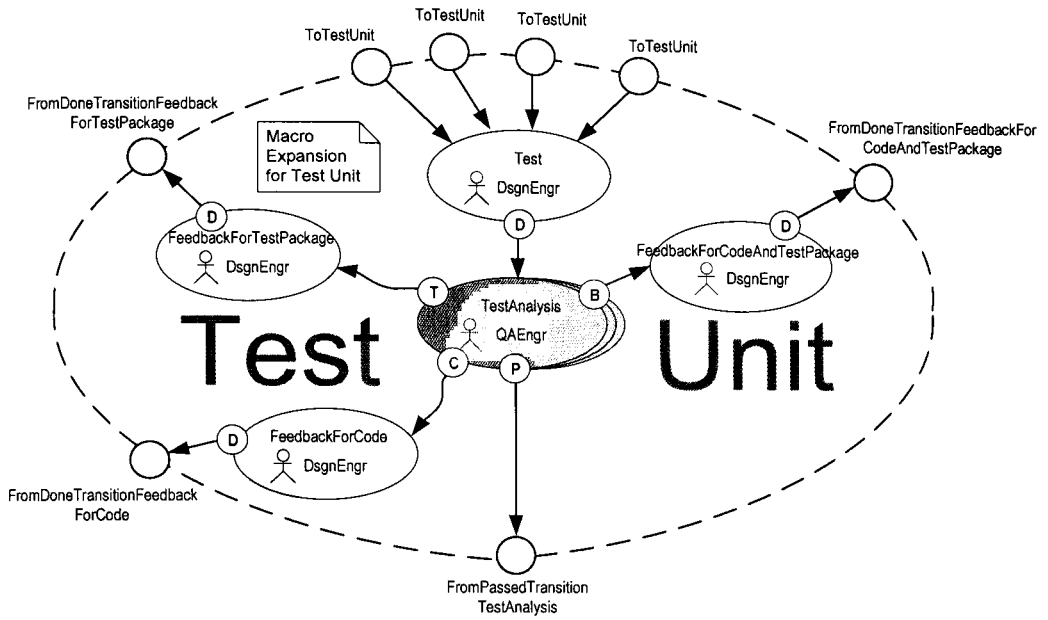


Figure 7-5 Amendment to a Macro Test Unit

It is felt that VRPML's replicator node could be dropped in order to reduce the complexity of a graph. The semantics of a replicator node could be assimilated by extending the semantics of an arc, that is, by permitting 1:M arcs [51]. An example use of the replicator node can be seen in Figure 7-6 alongside the proposed 1:M arc.

Figure 7-6 Assimilating Replicator Node with 1:M arc

Another possibility for simplifying the notation would be to eliminate the general purpose activity node. Obviously, a multi-instance activity node could straightforwardly be used in place of a general purpose activity node by specifying the depth of that multi-instance activity node to be 1. In fact, the advantage of employing a multi-instance activity node over a general purpose activity node is that no prior assumption need be made about the actual number of software engineers needed to complete that activity. Furthermore, as demonstrated by the

VRPML solution to the waterfall development model in Chapter 5, multi-instance and meeting activity nodes were sufficient to construct that process model. Nevertheless, eliminating the general purpose activity node from the notation can also be disadvantageous. For example, as far as readability of a VRPML graph is concerned, it can be difficult to distinguish whether an activity will be solely performed by one person or collaboratively by more than one person. Therefore, it is suggested that both general purpose activity nodes and multi-instance activity nodes are kept as part of the notation.

Another aspect that can be considered to improve the readability of a VRPML graph relates to the naming of transitions. Rather than allowing only a single letter as a way to identify transitions associated with a node, more meaningful names could be used. This is especially important if a particular node has many associated transitions. As an illustration, Figure 7-7 depicts the activity Test Analysis employing more meaningful names for the transitions.

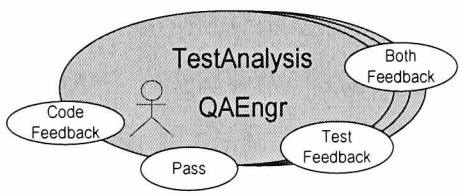


Figure 7-7 Example Amendments to Naming of Transitions

Finally, as discussed earlier, VRPML suffers from the problem of scale. Even for a relatively small problem such as the ISPW-6, VRPML requires two process models. Each process model requires twelve different activity nodes (including the macro expansions) as well as twelve corresponding workspaces. Apart from reuse, there seems to be no solution to this problem as yet although one possibility might be to investigate the use of a three dimensional space.

Expressiveness

To assess how well VRPML supports the modelling and enacting software processes, it is necessary to evaluate the expressiveness of its notation. In Chapters 5 and 6, the modelling of the ISPW-6 problem and the software processes based on

the waterfall model have been demonstrated. Furthermore, enactment has been demonstrated for the ISPW-6 problem. Given that the ISPW-6 problem has been formulated by experts in the field of software engineering, it should contain different types of (subtle) process issues seen in the real world. Thus, the fact that VRPML was able to support its modelling and enactment is a positive indication of the expressiveness of the VRPML notation.

Nevertheless, the fact that VRPML is based on the control-flow model raises an issue relating to the problem of *race conditions*, whereby two or more control flow signals compete to enable a particular activity node. Figure 7-8 illustrates the situation.

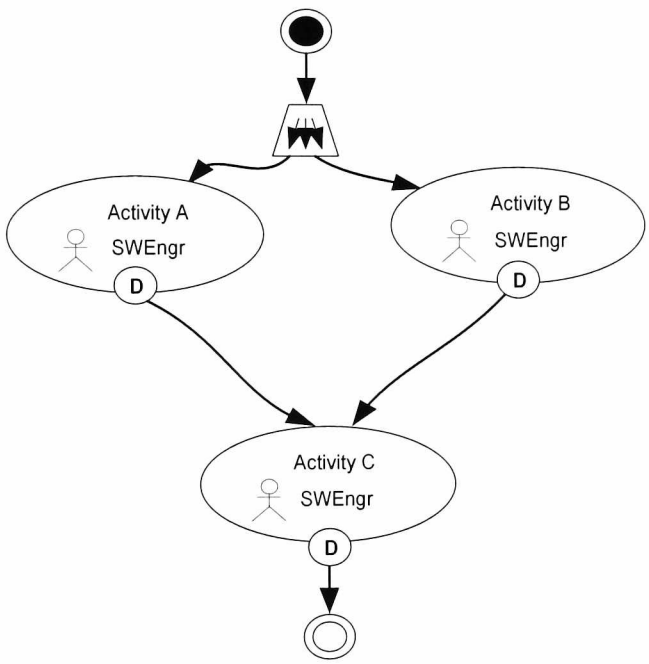


Figure 7-8 The Problem of Race Conditions

Although syntactically correct, the VRPML graph above represents a peculiar situation where activity C can be enabled when either the done transition of activity A or the done transition of activity B is selected. As a result, there is a possibility for activity C to be erroneously enabled twice.

One solution might be to include a merger node connecting arcs from the done transition from both activities A and B as depicted in Figure 7-9 below.

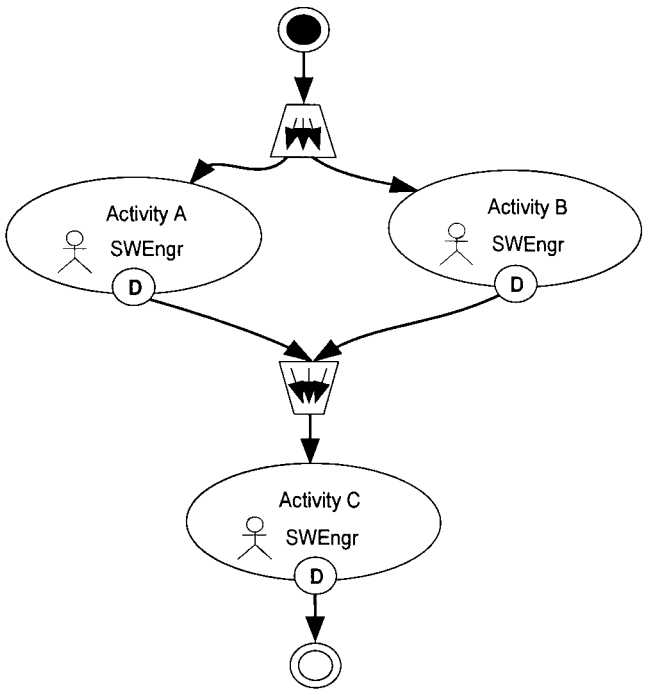


Figure 7-9 Avoiding Race Conditions Using a Merger Node

Here, no race conditions will occur as activity C can only be enabled once when both the done transitions of activity A and B have been selected. Although the inclusion of merger node solves the race condition problems associated with the above graph, this solution is not completely satisfactory because it does not prevent the process engineer from (unintentionally) introducing race conditions in the first place. Therefore, it is felt that a more general solution is required in VRPML to effectively handle race conditions.

Instead of permitting multiple incoming arcs, the syntax of an activity node could be changed to allow only a single incoming arc to connect to it. Consequently, referring to Figure 7-8, the two arcs connections from the done transitions of activities A and B to activity C would be syntactically incorrect. In this way, the possibilities for race conditions would be eliminated.

This proposed change of syntax to activity nodes raises a subtle issue relating to iterations and "feedback loops". Often, iterations and feedback loops need to be captured as part of the process model, for example, to specify repeated modification of a design until the design is approved. If only a single arc is permitted to connect

to an activity node, feedback loops to a previous activity node are no longer possible (i.e. VRPML graphs are now acyclic). As an illustration, the VRPML graph in Figure 7-10 below would give rise to a syntax error during "compilation" due to the more than one arc connection to activity P (i.e. from the start node and the transition Redo from activity Q).

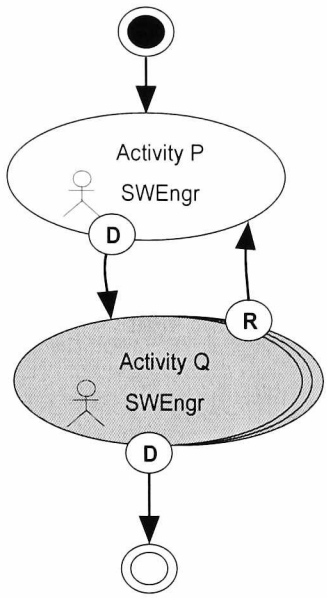


Figure 7-10 Example of Feedback Loop in VRPML

As discussed above, the reason for such feedback loops in VRPML is to support the idea that an activity might not have produced acceptable results and hence has to be repeated in some manner. This is such a fundamental aspect of software processes that it is an inherent part of the standard VRPML activity node, represented by decomposable transitions and re-enabled nodes. The graph shown in Figure 7-10 can be viewed as a generalisation of this situation, where activity Q is viewed as the post-condition check on activity P. (Of course, in this simple graph activity Q could have been represented as a decomposable transition on activity P, but the general case still needs consideration). To permit the general case of a feedback loop in a VRPML graph a new *decomposable cyclic node* could be introduced. Figure 7-11 depicts the proposed notation for the decomposable cyclic node.

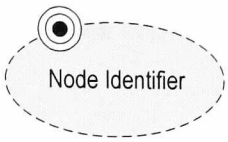


Figure 7-11 The Proposed Decomposable Cyclic Node Notation

Semantically, a decomposable cyclic node permits the specification of sub-graphs and allows the use of re-enable nodes to re-enable the parent of those sub-graphs (i.e. the decomposable cyclic node itself). In this way, general feedback loops can be specified.

An example usage of the decomposable cyclic node called "Activity P and Q" alongside its decomposition is demonstrated below to express the feedback loop for the VRPML graph given earlier (see Figure 7-10).

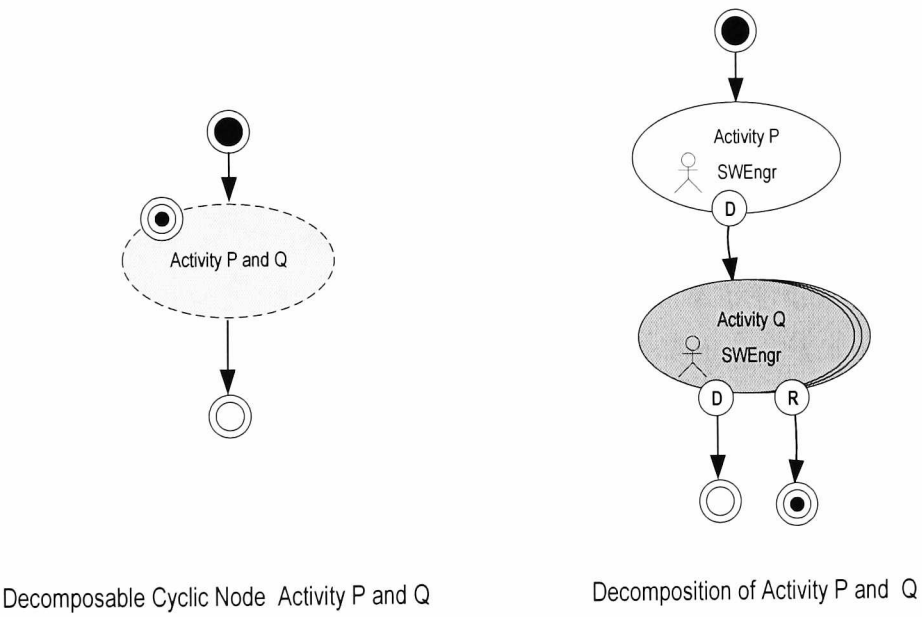


Figure 7-12 Example Usage of the Proposed Decomposable Cyclic Node

Referring to Figure 7-12, when the activity Q fails, the assigned software engineer can select the transition R (for Redo). As a result, a control-flow signal will be generated to enable its parent node (i.e. Activity P and Q) through a re-enabled node (shown as two white circles enclosing black circle). Otherwise, if the activity Q is successful, the assigned engineer can select the transition D (for Done). In this

case, the control-flow will be generated and propagated back to the main graph to enable the connected node.

To further investigate whether or not the proposed decomposable cyclic nodes is practicable to support feedback loops as well as to remove the possibilities for race conditions, Figure 7-13 revisits the ISPW-6 problem by applying that node as part of the VRPML solution.

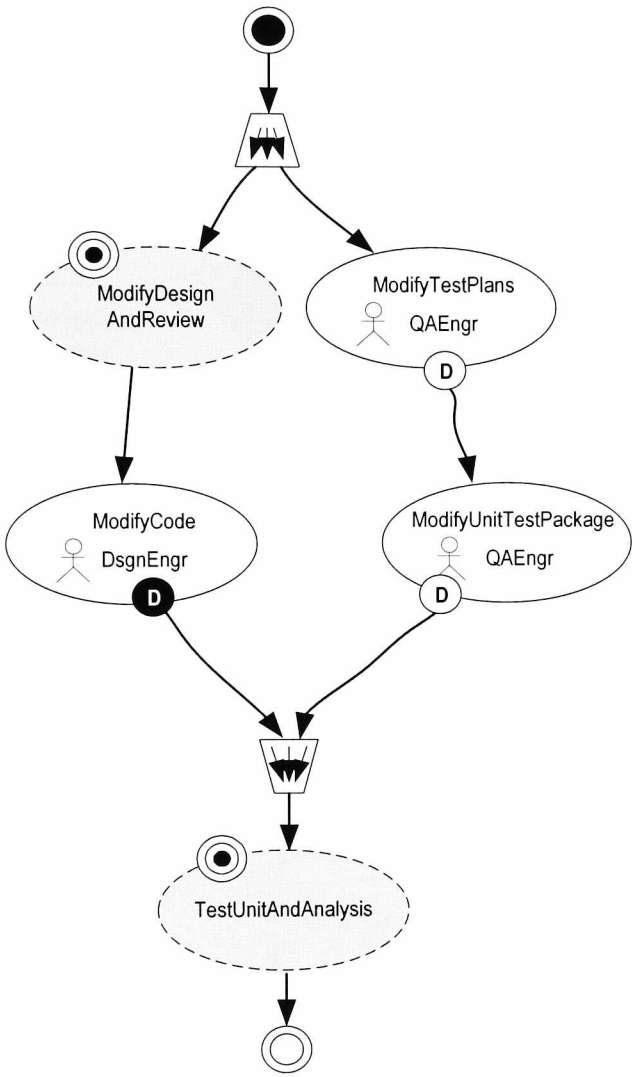


Figure 7-13 Using the Decomposable Cyclic Node as part of the VRPML Solution to the ISPW-6 Problem

Two decomposable cyclic nodes can be used:

i. Modify Design and Review

ii. Test Unit and Analysis

Because the decomposition for Modify Design and Review is relatively straightforward and similar to the case discussed earlier in Figure 7-12, it will not be developed further. Instead, only the decomposition for Test Unit and Analysis will be shown in Figure 7-14 below.

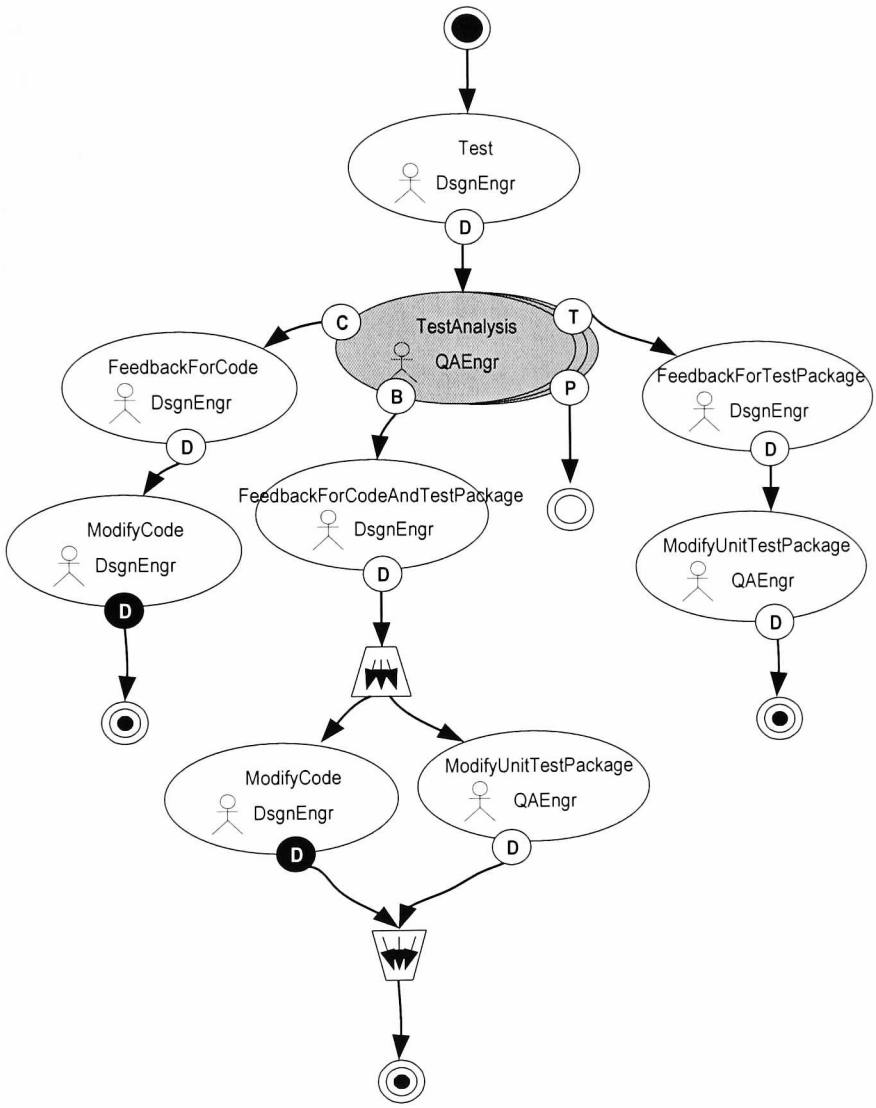


Figure 7-14 Decomposition of Test Unit and Analysis

The fact that multiple re-enabled nodes may be required to implement the decomposition cyclic node raises the possibility of race conditions. In effect, more than one re-enabled node could compete to re-enable the same parent node, although such situation may not occur in a properly formed VRPML graph such as

the one shown above. To ensure that race conditions would never cause problems, the semantics of a re-enabled node could be changed so that when one fires, an exception could be thrown if there are running activities within the scope of the decomposition. In this way, the process engineer could have the flexibility to decide whether to abort or to allow those activities to continue. Nevertheless, VRPML needs to provide a mechanism allowing the process engineer to manually raise a resource exception for a currently running activity in order to permit its abortion.

Finally, although not illustrated in the graph above, there is also a need to change the semantics of a stop node in order to address race conditions caused by an ill-formed VRPML graph. Similar to the semantics of a re-enabled node, when a stop node is reached, an exception could be thrown if there are running activities within the scope of the decomposition. The process engineer should then decide whether or not to abort those activities (by utilising the mechanism described above). If a stop node is used in the main VRPML graph, reaching a stop node aborts the overall enactment.

7.1.2 Computational Model

Although a computational model based on control-flow was initially chosen instead of a data-flow or a combined model (see Chapter 3), the author's experience from developing VRPML reveals the need to employ both control-flow and data-flow semantics in order to support the modelling and enacting of software processes. This is because the concepts of control-flow and data-flow are interrelated: the program control as well as the execution of an activity depends on the availability of data, and the flow of data is often governed by some notion of control. While having control-flow firing rule at the graph level, the VRPML enactment model also relies on data dependency semantics in order to enact a particular activity. Therefore, VRPML can be viewed as a PML which adopts a combined model similar to Little JIL, although the flow of data does not appear directly as part of the process graphs. As discussed earlier, the fact that the flow of data does not appear directly as part of process graphs is significant to reduce the visual complexity of the process models expressed in VRPML.

Demonstrated by the description of the ISPW-6 problem in Chapter 5 and the subsequent model expressed in VRPML, software processes tend to be imperative in nature. Therefore, the computational model based on control-flow at the graph level appears to be appropriate to express such behaviour.

As the VRPML firing rule at the graph level is based on control-flow, the author's experience in the modelling of activities within the waterfall development model showed the need to manually analyse the input and output dependencies in order to properly sequence those activities. If VRPML were purely based on the data-flow model, the sequencing of activities would be achieved automatically based on the availability of data at runtime. In fact, parallelism among activities can be opportunistically achieved in the same manner. Thus, in this respect, the pure data-flow model can be helpful to facilitate the construction of process models as the process engineers need not be concerned about parallelism.

Because activities specified in a process model are often performed by many engineers, there is a need for a PML to address issues relating to artefact sharing. In VRPML, artefact sharing can be flexibly achieved in the sense that activities could be tailored to deal with shared or copies of data. Obviously, there are pros and cons for either choice. If the process engineer chooses to allocate resources based on shared data, VRPML ensures that data consistency is maintained via access control mechanisms associated with artefacts. However, the disadvantage of working with shared data is that parallelism may be restricted due to runtime data "locking" imposed by access controls. If the process engineer chooses to allocate resources based on copies of data (i.e. like most pure data-flow models would), parallelism may be maximised. Nonetheless, there is a need to utilise a special tool such as the Concurrent Versions System [1] in order to ensure that data consistency is maintained if changes to copies of data have to be merged together.

Overall, as the above discussion illustrates, the VRPML computational model usefully balances control-flow firing rule and data dependency semantics without adding visual complexity to the language. For this reason, the VRPML computational model seems appropriate for supporting the modelling and enacting of software processes. However, as follow-on research, redesigning VRPML and

adopting the data-flow model whilst subjecting it to the same case study problems might be helpful in order to allow further evaluation and in-depth comparison.

7.1.3 Novel Language Claims

This section focuses on the evaluation of the VRPML novel language features as demonstrated throughout Chapter 5 and Chapter 6. These features include: the support for dynamic allocation of resources, and the support for awareness and visualisation issues.

Dynamic allocation of resources

The experiment in Chapter 6 has demonstrated the support for dynamic allocation of resources in VRPML, that is, by exploiting the enactment model and the resource exception handling mechanism. From the experiment, it can be seen that such support removes the need to utilise evolution mechanisms in order to cope with the dynamic changes needed because of the availability of resources. As discussed in Chapter 2, while enabling support for dynamic evolution is a desirable feature of a PML (e.g. through reflection), employing it to address dynamic allocation of resources can be expensive and impractical. Thus, the fact that VRPML supports dynamic allocation of resources without resorting to evolution improves its applicability for modelling and enacting software processes.

Concerning its resource exception handling mechanism, VRPML adopts the same approach as Little JIL [66] in the sense that resource exceptions are thrown when assigned resources are unavailable or could not be acquired. There is, however, a subtle difference between the resource exception handling mechanism in VRPML and Little JIL. In Little JIL, the handler for every resource exception must be defined. This is achieved through the handler badge. When no handler for a particular resource exception is defined, enactment of the activity in question will be automatically terminated. Even if the handler for a particular exception is defined, Little JIL limits the outcome of the activity whose exception is raised to retracting (for potentially re-posting) or terminating. Little JIL does not allow the process engineer to intervene, handle the exception, and resume the activity. In contrast, resource exceptions are handled in VRPML without the need to define the

handler. This is achieved by suspending enactment and creating specific tasks for the process engineers to rectify the resource exception and allow enactment to resume. As discussed in Chapter 4, VRPML also allows the process engineer to abort the activity completely and achieve the same effect as termination in Little JIL. Despite providing a useful resource exception handling mechanism, Little JIL falls short in not being able to suspend enactment and allow rectification through human intervention as in VRPML. Therefore, the same support for dynamic allocation of resources may not be achieved in Little JIL.

Awareness and visualisation support

The experiment in Chapter 6 has demonstrated that the integration with a virtual environment at the VRPML enactment level is possible, and hence a virtual environment is available for use. Therefore, the main issue to be considered is whether or not the awareness and visualisation issues can be better supported as a result.

In term of awareness support for the software engineers undertaking their assigned activities, the workspace defines the precise context for performing the activity in terms of artefacts to manipulate and tools to use. Furthermore, the workspace for each type of activity node can be straightforwardly distinguished during enactment. Such awareness is useful to encourage inter-person communication especially if a particular activity involves more than one person. As demonstrated in Chapter 6, this can be achieved, for example, by varying the graphical representation as well as incorporating artefacts and tools according to an activity's work context. Reiterating the discussion in Chapter 2, there is actually much more to awareness than being conscious about one's own work context and knowing and communicating with other engineers (if any) doing the same activity. Although useful by giving focus to the working context of a particular activity, it is felt that workspaces provide insufficient coverage in terms of the overall context of an activity in relation to other activities during enactment. Therefore, awareness issues concerning the dependencies among activities in a software process have still not been addressed, although the VRPML graphs could be a starting point as will be discussed in Chapter 8.

In demonstrating enactment from the experiment, it was noted that the rectification of a resource exception in a way can be viewed as a step in a meta-process. Thus, being an activity in itself, the rectification of resource exceptions would also benefit from the support for awareness just like other activities in a software process. Therefore, as an improvement, it follows that VRPML could provide a unique workspace for supporting the rectification of resource exceptions. One possibility is to include the list of available engineers, tools and artefacts available for selection in such a workspace.

Comparatively, the concept of workspaces in VRPML are somewhat similar to those of Merlin [44] and APEL [24]. Although resources can be dynamically allocated, VRPML workspaces are static, that is, they are fixed by the work contexts as envisioned by the process engineers. Merlin, on the other hand, has dynamic workspaces which are automatically generated by the runtime engine, based on the forward and backward chaining mechanisms traversing the Merlin's Prolog-based rules. Being a rule-based PML, the support for dynamic workspaces in Merlin is potentially advantageous because the process engineers can concentrate more constructing the rules, as they are relieved from task of specifying the work contexts in order to model a software process. Nevertheless, despite being dynamic, Merlin's workspaces are somewhat difficult to comprehend as they are built in the form of simplified entity relationship diagrams which software engineers need to interact with in order to perform their assigned activities. Although not completely implemented in the current prototype, workspaces in VRPML can be made more intuitive, for example, by representing artefacts and tools using familiar real-life icons in a virtual environment.

In APEL, activities and their sub-activities, as well as the flow of controls and artefacts, are shown to the software engineer during enactment through the desktop paradigm. In addition, similar to Merlin, the software engineer may also interact with the desktop paradigm to perform the activity. Although useful by giving the overall dependencies amongst activities in a software process, the desktop paradigm lacks focus particularly on the working context of an activity. Thus, the concept of workspaces in VRPML and the desktop paradigm could be usefully exploited to complement each other.

As far visualisation is concerned, the support provided by VRPML is still limited. Although workspaces and the access rights of artefacts are distinguished, artefacts and tools in a workspace simply map one-to-one to objects in a virtual environment. Therefore, in order to fully exploit the potential of the integration with a virtual environment, other types of mapping could be explored to permit a software engineer to dynamically "switch" between different mappings in order to enhance visualisation support. This issue will be further discussed in the scope of future work in Chapter 8.

7.2 Summary

This chapter has presented an evaluation of VRPML in terms of the language syntax and semantics, the choice of computational model, and the novel language claims. This evaluation has uncovered strengths and limitations of VRPML and some improvements to the language have been suggested. Although VRPML has some characteristics which might be considered unnecessary, it is believed that the benefits from the use of VRPML outweigh its drawbacks.

In the final chapter, work presented in this thesis is drawn together, summarising the research undertaken and discussing the impact of the results achieved. Discussions are offered to evaluate the main research hypothesis, that is, whether or not VRPML is useful for supporting distributed software engineering teams, alongside conclusions and suggestions for future work.

Chapter 8 – Conclusion

The aim of this research work was to develop and evaluate a new visual and enactable PML, VRPML, which addresses some of the perceived deficiencies in other PMLs – particularly in terms of the support for the human dimension issues as well as the support for dynamic allocation of resources. In the context of this aim, this chapter summarises the important points arising from the earlier chapters in order to discuss the impacts of the results achieved and the implication for future work.

8.1 Overview

Chapter 1 set the scene by establishing the need to model a software process using a PML. Through process enactment, a key feature of a PML, process models are able to provide automation, guidance, and enforcement of software engineering practices.

A key point was raised in Chapter 2 that despite much fruitful research into PMLs, their takeup by industry has not been widespread. It was therefore necessary to identify areas where progress has been made, and areas where potential improvements could be achieved. Chapter 2 presented an account of the state-of-the-art of PMLs by discussing a classification and a taxonomy for PMLs. Using the taxonomy, Chapter 2 provided a survey of existing PMLs, and revealed a number of perceived deficiencies. In the scope of this work, some deficiencies relating to the support for human dimension issues (e.g. awareness and visualisation), and the support for dynamic allocation of resources were considered in line with the overall context of supporting distributed software engineering teams. As a result, the development of a new visual PML, VRPML, was proposed. VRPML is a visual PML which considers the virtual environment as a fundamental component, manipulatable as part of the construction of the process model (i.e. via features in the language) as well as being part of the runtime environment, and supports dynamic allocation of resources through its enactment model.

Chapter 3 was dedicated to the design issues for VRPML. The first issue was whether or not it is useful to exploit a virtual environment as a way of supporting a software process. The second issue concerned flow-based visual languages as a basis for VRPML. In particular, two possible computational models for VRPML were debated: the control-flow model and the data-flow model. Based on the author's initial evaluation, it was decided that the control-flow model was more suitable for VRPML process model graphs. Among the main reasons were that the control-flow model facilitates reasoning about execution semantics and tracing enactment as well as supporting cyclic behaviour in a straightforward manner. Finally, a brief discussion of a novel enactment model supporting the dynamic allocation of resources was outlined.

Chapter 4 presented the proposed design notation for VRPML. A partial solution of the ISPW-6 problem was used as a way of investigating the syntax and semantics of VRPML. Additionally, the enactment model for VRPML was also described.

The work in Chapter 5 outlined the modelling of the two case study problems: the ISPW-6 problem and the waterfall development model. The main aim of undertaking the two case study problems was to demonstrate that VRPML has a sufficiently rich notation to support modelling and enacting of software processes. Part of the aim relating to the modelling aspect was achieved, that is, it was shown that VRPML notation was successful in providing a sound solution to both problems. The other part of the aim concerning enactment was considered in Chapter 6.

8.2 Results

The aim of Chapter 6 was to investigate whether a process model expressed in VRPML can actually be enacted and whether enactment indentifies further issues for consideration. Therefore, the chapter concentrated on the issues relating to implementation. The components of runtime support system and their expected functionalities were discussed, and several of them were actually built as prototypes in this work, including:

- Runtime Interpreter
- Runtime Client

- To-do-list Manager
- Workspace Manager

Although far from being complete, these prototype components were helpful to demonstrate that the complete support environment could actually be built for VRPML.

To investigate the novel language claims, the main aspects of VRPML considered for exploration were:

- Support for enactment in a distributed environment
- Support for dynamic creation of tasks and allocation of resources by exploiting the enactment model
- Support for integration with a virtual environment at the PML enactment level so that awareness and visualisation issues can be addressed

One experiment involving enactment of the VRPML solution to the ISPW-6 was described in order to validate the aforementioned aspects, and hence show that VRPML syntax and semantic are sound. In particular, the experiment demonstrated the VRPML enactment model and its resource exception handling mechanism along with the support for enactment in a distributed environment and the support for dynamic creation of task and allocation of resources.

Additionally, the experiment also showed the feasibility of supporting awareness and visualisation issues. This was achieved by demonstrating the integration of a virtual environment at the PML enactment level. Overall, the experiment described in Chapter 6 was considered successful as it demonstrated the key features of VRPML and provided practical evidence of how these features are actually being used in the context of supporting the modelling and enacting of software processes.

Finally, Chapter 7 was devoted to the evaluation of VRPML. The VRPML solution to ISPW-6 problem and the waterfall model were revisited and evaluated. The VRPML solution to the ISPW-6 problem was also compared to Little JIL particularly in terms of the effectiveness of representation and expressiveness. Furthermore, some improvements to VRPML were introduced. The author's initial choice of control-flow as the main computational model for VRPML was also

evaluated along with the novel language features. Overall, it was found that although VRPML suffers from some limitations, its advantages outweigh the disadvantages.

8.3 Discussion

In line with the main research hypothesis discussed in Chapter 1, this section debates the applicability of VRPML for supporting distributed software engineering teams. In doing so, this section identifies some of the difficulties associated with distributed software processes, and analyses whether or not the features provided in VRPML addresses those difficulties.

Due to the lack of face-to-face contact, coordination of activities involved in a software process is often difficult when the development teams are not physically collocated. The fact that VRPML supports the construction of the process model as well as its enactment in a distributed environment is helpful in this situation. One reason is that the coordination of activities can be fully automated through enactment.

Another common problem arising from the lack of face-to-face contact relates to communication breakdown amongst the team members. Generally, communication breakdown has a negative effect on the developed software, resulting in bugs and unnecessary rework. As a consequence, the probability of development project success can be significantly reduced. Although not fully implemented in the current prototype, the support for awareness in VRPML may be helpful to address some of these issues. This is because through awareness, group cohesion may be improved, and hence encourage informal communication amongst team members.

Nevertheless, communication amongst team members can often be difficult when the development teams are distributed in multiple sites. Asynchronous communication tools (e.g. email tools) address this issue to a certain degree, but do not allow software engineers to hold the rich discussions possible when they are physically collocated. Thus, the feature of VRPML that permits the specification of synchronous communication tools (e.g. a tele-conferencing program) as part of the workspace definition can be helpful to address this issue. Furthermore, VRPML also provides a special node for virtual meetings. The support for virtual meetings

is beneficial since meetings are an important characteristic of software engineering. Additionally, when development teams are distributed over multiple sites, virtual meetings could help reduce costs if a meeting would otherwise have to be held face-to-face.

In the context of distributed software engineering teams affected by both geographical and temporal distribution, collaboration on a shared activity can also be difficult to achieve. This is because there may be only a small window of overlap in term of times when the team members can work together. In some cases, there could also be absolutely no window of overlap at all. The fact that VRPML permits dynamic allocation of resources might be convenient to address some of the above issues. One reason is that the assignment of engineers as resources to a shared activity can be made dynamically not only based on the availability of engineers but also depending on whether or not there is a temporal overlap for team members to collaborate.

As has been shown, some of the features of VRPML can be usefully exploited to address some of the problems associated with distributed software processes. Therefore, it can be concluded that VRPML is useful for supporting software processes for distributed software engineering teams.

8.4 Future Work

As the current implementation of VRPML is still in a prototype form, an obvious starting point for future work would be to complete the implementation. For example, implementing a compiler and complete workspace manager would be a useful endeavour, together with support for the additional language features proposed in Chapter 7.

Referring to the analysis in Chapter 2, other obvious future work for VRPML is to support reflection and the collection of enactment data. As discussed in Chapter 2, with reflection, VRPML could support the dynamic evolution of the process model and the meta-process issues. Through the collection of enactment data, VRPML could support systematic and objective evaluation of a particular process model, for example, as indicators for facilitating process improvement.

As discussed in Chapter 4, the semantics of VRPML have been informally defined, and captured in the implementation of the run-time interpreter. Further work could provide a formal specification for VRPML to enable additional insight into the language and its characteristics to be derived.

Apart from the above, a number of other research avenues could also be investigated. The fact that VRPML supports integration with a virtual environment opens up many possibilities for using visualisation to provide multiple views of the same process model from different perspectives, and hence potentially improving process understanding. Instead of using a straightforward mapping of workspaces, that is, a workspace, artefacts, tools and task descriptions map one-to-one to a virtual room and objects, other meaningful visualisations could also be explored by defining or using other types of mapping. For example, artefact-centred mapping as defined by Doppke *et al* (discussed in Chapter 3) could be used where artefacts are represented as virtual rooms and their dependency relationships are expressed as part of the arrangement of the rooms. If sub-products of the artefacts are defined then they are represented as separate rooms connected to the parent product room either by exits or by containment, whilst tools and task description can be defined as objects inside the room. Clearly, by manipulating the types of mapping used, multiple views of the same process can be achieved. In turn, such views may enhance the support for awareness.

Another possible area of research is to find other ways of addressing awareness in VRPML, for example, supporting user awareness by representing software engineers as avatars in the workspaces during enactment or using live video. Using avatars or live video can perhaps improve the sense of realism and further encourage informal communication as engineers can "see" each other. This is especially useful if the workspaces involve more than one person, and the software engineering teams are physically distributed.

Although useful by giving focus on a particular activity, the evaluation in Chapter 7 establishes that workspaces defined by VRPML give insufficient working context particularly about the overall activities, that is, in terms of how the pieces fit together into the whole picture. In the current VRPML implementation, questions such as what the previous task was, what the next task is, and what needs to be

done to move along cannot be easily answered. Therefore, it would be useful to find ways for VRPML to also give context of the overall activity, for example, by giving the software engineers access to the enacted VRPML graph in the forms of animated flow of control during enactment.

Because a software development project often involves hard deadlines, another possibility for further work is to investigate the inclusion of "timing" criteria as part of the VRML notation. Perhaps, the timing criteria could be exploited as part of the workspace definitions, raising an "alarm" when an activity is due to be completed. As a result, the software engineers can be reminded about the deadlines of the activities that they are undertaking.

Finally, as VRPML is a domain-specific language for modelling and enacting of software processes, a more general suggestion for future work is to investigate the applicability and usefulness of VRPML to support general workflow processes.

8.5 Closing Remarks

Because of the potential benefits in terms of being able to provide automation, guidance and enforcement of software engineering practices and policies through enactment, a PML could form an important feature of future software engineering environments. Ironically, despite the above promise, the adoption of PMLs in industry has not been widespread. Furthermore, no single existing PML has emerged dominant as the *de facto* standard for modelling and enacting software processes. For these reasons, it follows that research into PMLs is still necessary.

In the context of the work reported in this thesis, despite some limitations, VRPML represents another step forward in the research into PMLs, particularly as it addresses the dynamic management of resource issues as well as opens up new possibilities of supporting awareness and visualisation issues in software processes by exploiting a virtual environment. Based on the experience and the subsequent evaluation of this research work, the research hypothesis discussed in Chapter 1 has been proved – integration with a virtual environment in VRPML is indeed useful for supporting software processes for distributed software engineering teams, and VRPML provides a significant improvement towards the goal of a practicable PML.

References

1. CVS, Concurrent Versions System, http://www.cvshome.org, 2003.
2. Ackerman, W.B. Data Flow Languages *IEEE Computer*, 1982, 15-23.
3. Agerwala, T. and Arvind. Data Flow Systems *IEEE Computer*, 1982, 10-13.
4. Ambriola, V., Ciancarini, P. and Montangero, C., Software Process Enactment in OIKOS. in *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '90)*, (1990), ACM Press, 183-192.
5. Ambriola, V., Conradi, R. and Fuggetta, A. Assessing Process-Centered Software Engineering Environments. *ACM Transactions on Software Engineering and Methodology*, *6* (3). 283-328.
6. Arbaoui, S., Lonchamp, J. and Montangero, C. The Human Dimension of the Software Process. in Derniame, J.C., Kaba, B.A. and Wastell, D. eds. *Software Process: Principles, Methodology and Technology*, Lecture Notes in Computer Science Volume 1500, Springer, Berlin-Heidelberg, 1999, 165-196.
7. Baldi, M., Gai, S., Jaccheri, M.L. and Lago, P. Object-Oriented Software Process Model Design in E3. in Finkelstein, A., Kramer, J. and Nuseibeh, B. eds. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 279-290.
8. Bandinelli, S., Fuggetta, A. and Ghezzi, C. Software Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering*, *19* (12). 1128-1144.
9. Bandinelli, S., Fuggetta, A., Ghezzi, C. and Lavazza, L. SPADE: An Environment for Software Process Analysis, Design and Enactment. in Finkelstein, A., Kramer, J. and Nuseibeh, B. eds. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 223-247.
10. Basili, V.R. and Rombach, H.D. The TAME Approach: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering*, *14* (6). 758-773.
11. Becattini, F., Nitto, E.D., Fuggetta, A. and Valetto, G., Exploiting MOOs to Provide Multiple Views for Software Process Support. in *Proceedings of International Process Technology Workshop*, (Villard de Lans - Genoble, France, September 1999).
12. Belkhatir, N., Estublier, J. and Melo, W. ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. in Nuseibeh, B. ed. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 187-222.
13. Bruynooghe, F., Greenwood, R.M., Robertson, I., Sa, J. and Warboys, B.C. PADM: Towards a Total Process Modeling System. in Finkelstein, A., Kramer, J. and Nuseibeh, B. eds. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 293-334.
14. Canals, G., Boudjlida, N., Derniame, J.C., Godart, C. and Lonchamp, J. ALF: A Framework for Building Process-Centred Software Engineering Environments. in Finkelstein, A., Kramer, J. and Nuseibeh, B. eds. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 153-185.
15. Chen, J.J. CSPL: An Ada95-Like, Unix-Based Process Environment. *IEEE Transactions on Software Engineering*, *23* (3). 171-184.
16. Chou, S.C. and Chen, J.J. Process Evolution Support in Concurrent Software Process Language Environment. *Information and Software Technology*, *41* (8). 507-524.
17. Ciancarini, P. and Rossi, D. Jada: A Coordination Toolkit for Java - Technical Report UBLCS-96-15, Department of Computer Science, University of Bologna, Italy, 1997.
18. Conradi, R., Fernstrom, C. and Fuggetta, A. A Conceptual Framework for Evolving Software Processes. *ACM SIGSOFT Software Engineering Notes*, *18* (4). 26-35.

19. Conradi, R., Fernstrom, C., Fuggetta, A. and Snowdon, R., Towards a Refence Framework for Process Concepts. in *Proceedings of the 2nd European Workshop on Software Process Technology*, (Trondheim, Norway, September 1992), Lecture Notes in Computer Science Volume 635, Springer, 3-17.
20. Conradi, R., Hagaseth, M., Larsen, J., Nguyen, M.N., Munch, B.P., Westby, P.H., Zhu, W., Jaccheri, M.L. and Liu, C. EPOS: Object-Oriented Cooperative Process Modelling. in Finkelstein, A., Kramer, J. and Nuseibeh, B. eds. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 33-69.
21. Conradi, R. and Jaccheri, M.L. Process Modelling Languages. in Derniame, J.C., Kaba, B.A. and Wastell, D. eds. *Software Process: Principles, Methodology and Technology*, Lecture Notes in Computer Science Volume 1500, Springer, Berlin-Heidelberg, 1999, 27-52.
22. Cugola, G., Nitto, E.D., Ghezzi, C. and Mantione, M., How to Deal with Deviations during Process Model Enactment. in *Proceedings of the 17th International Conference on Software Engineering*, (Seattle, Washington, April 1995), IEEE Computer Society Press, 265-273.
23. Dam, V.A. The Shape of Things to Come. *ACM SIGGRAPH Computer Graphics, 31* (1). 42-44.
24. Dami, S., Estublier, J. and Amiour, M. APEL: A Graphical Yet Executable Formalism for Process Modelling. *Automated Software Engineering, 5* (1). 61-96.
25. DeBellis, M. and Haapala, C. User-Centric Software Engineering *IEEE Expert*, 1995, 34-41.
26. Derniame, J.C., Kaba, B.A. and Warboys, B.C. The Software Process: Modelling and Technology. in Derniame, J.C., Kaba, B.A. and Wastell, D. eds. *Software Process: Principles, Methodology and Technology*, Lecture Notes in Computer Science Volume 1500, Springer, Berlin-Heidelberg, 1999, 1-13.
27. Doppke, J.C., Heimbigner, D. and Wolf, A.L. Software Process Modeling and Execution within Virtual Environments. *ACM Transactions on Software Engineering and Methodology, 7* (1). 1-40.
28. Dossick, S.E. and Kaiser, G., CHIME: A Metadata-Based Distributed Software Development Environments. in *Proceedings of the joint 7th European Software Engineering Conference and Foundation of Software Engineering (ESEC/FSE 99)*, (Toulouse, France, September 1999), Lecture Notes in Computer Science Volume 1687, Springer, 464-475.
29. Emmerich, W. and Como, V.G., FUNSOFT Nets: A Petri-Net based Software Process Modeling Language. in *Proceedings of the 6th International Workshop on Software Specification and Design*, (Italy, 1991), IEEE Computer Society Press, 175-184.
30. Engels, G. and Groenewegen, L. SOCCA: Specifications of Coordinated and Cooperative Activities. in Finkelstein, A., Kramer, J. and Nuseibeh, B. eds. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 71-102.
31. Franch, X. and Ribo, J.M., Using UML for Modelling the Static Part of a Software Process. in *In Proceedings of the 2nd Unified Modelling Language Conference (UML99)*, (Fort Collins, Colorado, 1999), Lecture Notes in Computer Science Volume 1723, Springer, 292-307.
32. Fuggetta, A., Software Process: A Roadmap. in *The Future of Software Engineering. In Conjunction with the 22nd International Conference on Software Engineering (ICSE 2000)*, (Limerick, Ireland, 2000), ACM Press, 27-34.
33. Gelernter, D. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems, 7* (1). 80-112.
34. Greenwood, R.M., Robertson, I., Snowdon, R.A. and Warboys, B.C., Active Models in Business. in *Proceedings the 5th. Conference on Business Information Technology CBIT '95.*, (Department of Business Information Technology, Manchester Metropolitan University, Manchester, November 1995).
35. Greenwood, R.M. and Warboys, B.C., ProcessWeb - Process Support for the World Wide Web. in *Proceedings of the 5th European Workshop (EWSPT '96)*, (Nancy, France, 1996), Lecture Notes in Computer Science Volume 1149, Springer, 82-85.

36. Grundy, J.C. and Hosking, J.G. Serendipity: Integrated Environment Support for Process Modeling, Enactment and Work Coordination. *Automated Software Engineering, 5* (1). 27-60.
37. Heiman, P., Joeris, G. and Krapp, C.A., DYNAMITE: Dynamic Task Nets for Software Process Management. in *Proceedings of the 18th International Conference on Software Engineering*, (Berlin, Germany, March 1996), IEEE Computer Prress, 331-341.
38. Huff, K.E. Software Process Modeling. in Fuggetta, A. and Wolf, A. eds. *Trends in Software Process*, John Wiley & Sons, 1996, 1-24.
39. Humphrey, W.S., People Consideration in Process Models. in *Proceedings of the 6th International Software Process Workshop*, (Hakodate, Hokkaido, Japan, 1990), IEEE Computer Society Press, 113-115.
40. Humphrey, W.S. and Kellner, M.I., Software Process Modeling: Principles of Entity Process Models. in *Proceedings of the 11th International Conference on Software Engineering*, (1989), IEEE Computer Society Press, 331-342.
41. Humprey, W.S. *Managing Software Process.* Addison Wesley, Reading, Mass., 1989.
42. Jaccheri, M.L., Conradi, R. and Drynes, B.H., Software Process Technology and Software Organisations. in *Proceedings of the 7th European Workshop on Software Process (EWSPT 2000)*, (Kaprun, Austria, February 2000), Lecture Notes in Computer Science Volume 1780, Springer, 96-108.
43. Jaccheri, M.L. and Stalhane, T., Evaluation of the E3 Process Modelling Language and Tool for the Purpose of Model Creation. in *Proceedings of the 3rd International Conference on Product Focused Software Process Improvements (PROFES01)*, (Kaiserslautern, Germany, 2001), Lecture Notes in Computer Science Volume 2188, Springer, 271-281.
44. Junkermann, G., Peuschel, B., Schafer, W. and Wolf, S. MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment. in Finkelstein, A., Kramer, J. and Nuseibeh, B. eds. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994, 103-129.
45. Kaiser, G.E. Experience with Process Modelling in the MARVEL Software Development Environment - Technical Report CUCS-446-89, Department of Computer Science, Columbia University, 1989.
46. Kaiser, G.E., Feiler, P.H. and Popovich, S. Intelligent Assistance for Software Development and Maintenance. *IEEE Software, 5* (3). 40-49.
47. Katayama, T., A Hierarchical and Functional Software Process Description and its Enaction. in *Proceedings of the 11th Conference on Software Engineering*, (Pittsburgh, Pennsylvania, May 1989), IEEE Computer Society Press, 343-352.
48. Kellner, M.I., Feiler, P.H., Finkelstein, A., Katayama, T., Osterweil, L.J., Penedo, M.H. and Rombach, H.D., Software Process Modeling Example Problem. in *Proceedings of the 6th International Software Process Workshop*, (Hakodate, Hokkaido, Japan, October 1990), IEEE Computer Society Press.
49. Konno, S. CyberVRML97 - Virtual Reality Modelling Language Development Library, 2002.
50. Lee, H. Evaluation of Little-JIL 1.0 with ISPW-6 Software Process Example. Technical Report UM-CS-1999-033, Department of Computer Science, University of Massachusetts at Amherst, March 1999.
51. Lee, P.A. and Webber, J. Taxonomy for Visual Parallel Programming Languages - Technical Report CS-TR-793, School of Computing Science, University of Newcastle upon Tyne, 2003.
52. Liu, C. and Conradi, R., Process Modeling Paradigms: An Evaluation. in *Proceedings of the 1st European Workshop on Software Process Modeling*, (Milano, Italy, May 1991), Italian National Association for Computer Science, 39-52.
53. Lonchamp, J. An Assessment Exercise. in Finkelstein, A., Kramer, J. and Nuseibeh, B. eds. *Software Process Modelling and Technology*, Research Studies Press Ltd., Taunton, Somerset, U.K., 1994, 335-356.
54. Melo, W.L., Belkhatir, N. and Estublier, J., User Modelling and Control in Adele System. in *Proceedings of the 4th International Conference on Computing and Information (ICCI 92)*, (1992), IEEE CS Press, 334-337.

55. Montangero, C. and Ambriola, V. OIKOS: Constructing Process-centred SDEs. in Finkelstein, A., Kramer, J. and Nuseibeh, B. eds. *Software Process Modelling and Technology*, Research Studies Press, Taunton, England, 1994.
56. Nguyen, M.N. and Conradi, R., Classification of Meta-Processes and their Models. in *Proceedings of the 3rd International Conference on Software Process (ICSP'3)*, (Washington, USA, 1994), IEEE CS Press, 167-175.
57. Ribo, J.M. and Franch, X. PROMENADE: A PML Intended to Enhance Standarization, Expressiveness and Modularity in Software Process Modelling. Research Report LSI-34-R., Llenguatges I Sistemes Informatics, Politechnical of Catalonia, Spain, 2000.
58. Royce, W.W., Managing the Development of Large Software Systems. in *Proceedings of IEEE WESCON*, (1970), 1-9.
59. Rumbaugh, J., Jacobson, I. and Booch, G. *The UML User Guide*. Addison Wesley, 1999.
60. Sommerville, I. *Software Engineering (Sixth Edition)*. Addison Wesley, 2001.
61. Sommerville, I. and Rodden, T. Human, Social and Organisational Influences on the Software Process. in Fuggetta, A. and Wolf, A. eds. *Trends in Software Process*, John Wiley & Sons, 1996, 89-108.
62. Sutton Jr., S., Heimbigner, D. and Osterweil, L.J. APPL/A: A Language for Software Process Programming. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *4* (3). 221-286.
63. Sutton Jr., S. and Osterweil, L.J., The Design of a Next-Generation Process Language. in *Proceedings of the Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, (1997), Lecture Notes in Computer Science Volume 1301, Springer, 142-158.
64. Swenson, K.D., A Visual Language to Describe Collaborative Work. in *Proceedings of the 1993 IEEE Symposium on Visual Language*, (1994), IEEE Computer Society Press, 298-303.
65. Warboys, B.C., The IPSE 2.5 Project: Process Modelling as a Basis for a Support Environment. in *Proceedings of the First International Conference on System Development Environments and Factories SDEF1*, (Berlin, 1989), Pitman Publishing.
66. Wise, A. Little JIL 1.0 Language Report - Technical Report 98-24, Department of Computer Science, University of Massachusetts at Amherst, April 1998.
67. Wise, A., Cass, A.G., Lerner, B.S., McCall, E.K., Osterweil, L.J. and Sutton Jr., S., Using Little JIL to Coordinate Agents in Software Engineering. in *Proceedings of the 15th International Conference on Automated Software Engineering*, (Grenoble, France, 2000), IEEE Computer Society Press, 155-164.
68. Yang, Y., Coordination for Process Support is Not Enough. in *Proceedings of the 4th European Workshop on Software Process Technology*, (1995), Lecture Notes in Computer Science Volume 913, Springer, 205-208.
69. Zamli, K.Z. Process Modeling Languages: A Literature Review. *Malaysia Journal of Computer Science*, *14* (2). 26-37.
70. Zamli, K.Z. and Lee, P.A., Exploiting a Virtual Environment in a Visual PML. in *Proceedings of the 4th International Conference on Product Focused Software Process Improvements (PROFES02)*, (Rovaniemi, Finland, 2002), Lecture Notes in Computer Science Volume 2559, Springer, 49-62.
71. Zamli, K.Z. and Lee, P.A., Modelling and Enacting Software Processes Using VRPML. in *Proceedings of the 10th IEEE Asia Pacific International Conference on Software Engineering (forthcoming December 2003)*, (Chiang Mai, Thailand, 2003), IEEE Computer Society Press.
72. Zamli, K.Z. and Lee, P.A., Taxonomy of Process Modeling Languages. in *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, (Beirut, Lebanon, June 2001), IEEE Computer Society Press, 435-437.

Appendix A: Non-Enactable PMLs

This appendix identifies a number of existing non-enactable PMLs. Some of these PMLs are actually being used in other domain, but have been applied in the context of supporting the modelling of software processes. Further details about any of these PMLs and their use can be obtained from the cited reference.

- SOCCA [30] – is a PML which uses the state transition diagram to support the modelling of software processes, and permits enactment of a process model to be simulated.
- E3 [7] – is an object-oriented PML employing graphical pre-defined classes and relations in order to support the modelling of software processes.
- LIMBO [55] – is a PML which serves as the specification language for PATE. Some characteristics of LIMBO are briefly described in Chapter 2.
- BM [13] – is a PML which serves as the specification language for PWI PML. Some characteristics of BM are briefly described in Chapter 2.
- STATEMATE [40] – is a PML which is based on state-charts (i.e. enhanced state transitions diagrams). It allows enactment of a process model to be simulated.
- UML [59] – is a graphical language for specifying object-oriented software design. Example usage of UML as a PML can be seen in the work by Franch and Ribo [31].
- ETVX – is a graphical notation developed by IBM in early 1980s to write work procedures. Example usage of ETVX as a PML can be seen in the work by Humprey [41].
- IDEF0 – is a graphical notation used to specify the work procedures based on the Structured Analysis and Design Techniques (SADT). Example usage of IDEF0 as a PML can be seen in the work by Jaccheri and Stalhane [43].

Appendix B: Complete VRPML Solution for the Waterfall Model

This appendix provides a complete solution for the waterfall model. Further discussion about the solution can be obtained from the main texts.

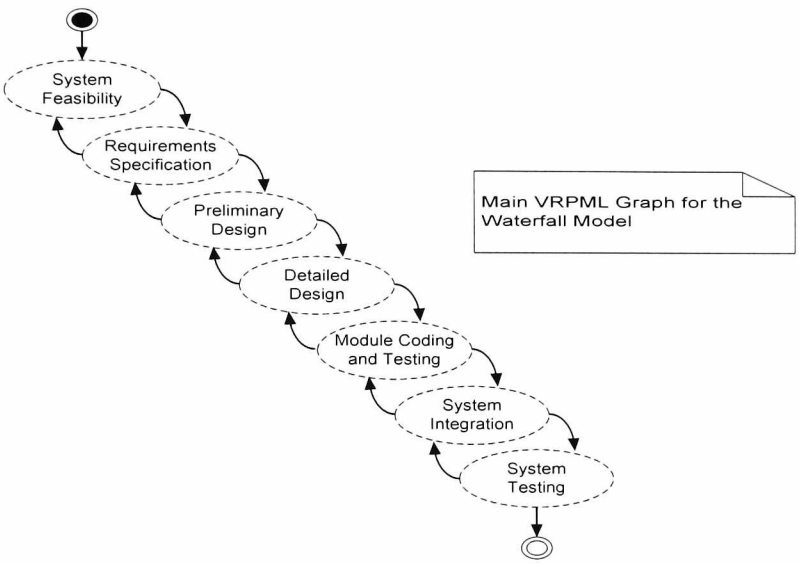


Figure B-1 The Main VRPML Graph for the Waterfall Development Model

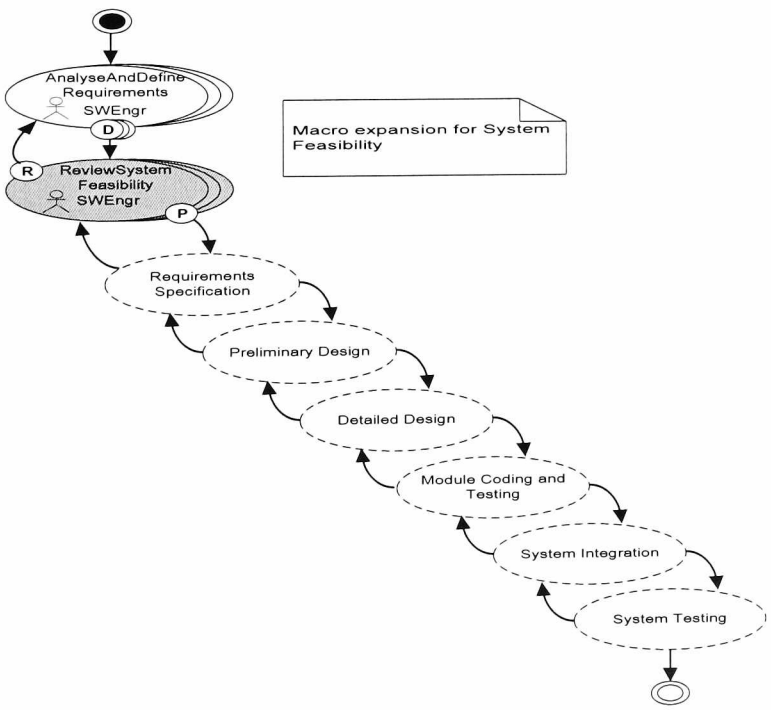


Figure B-2 Macro Expansion for System Feasiblity

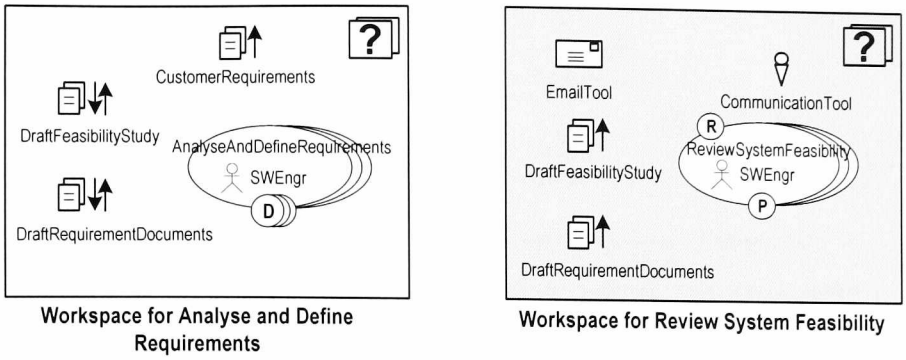


Figure B-3 Workspaces for Analyse and Define Requirements, and Review System Feasibility

Figure B-4 Macro Expansion for Requirement Specification

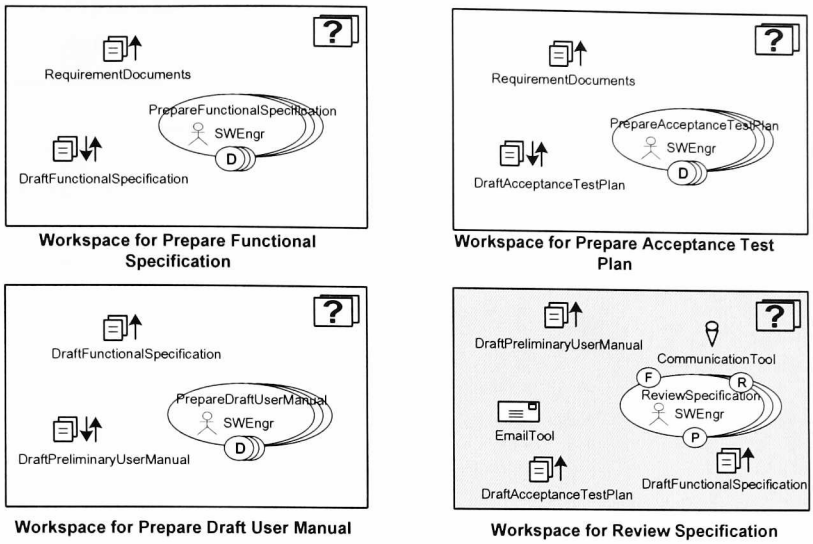


Figure B-5 Workspaces for Prepare Functional Specification, Prepare Acceptance Test Plan, Prepare Draft User Manual, and Review Specification

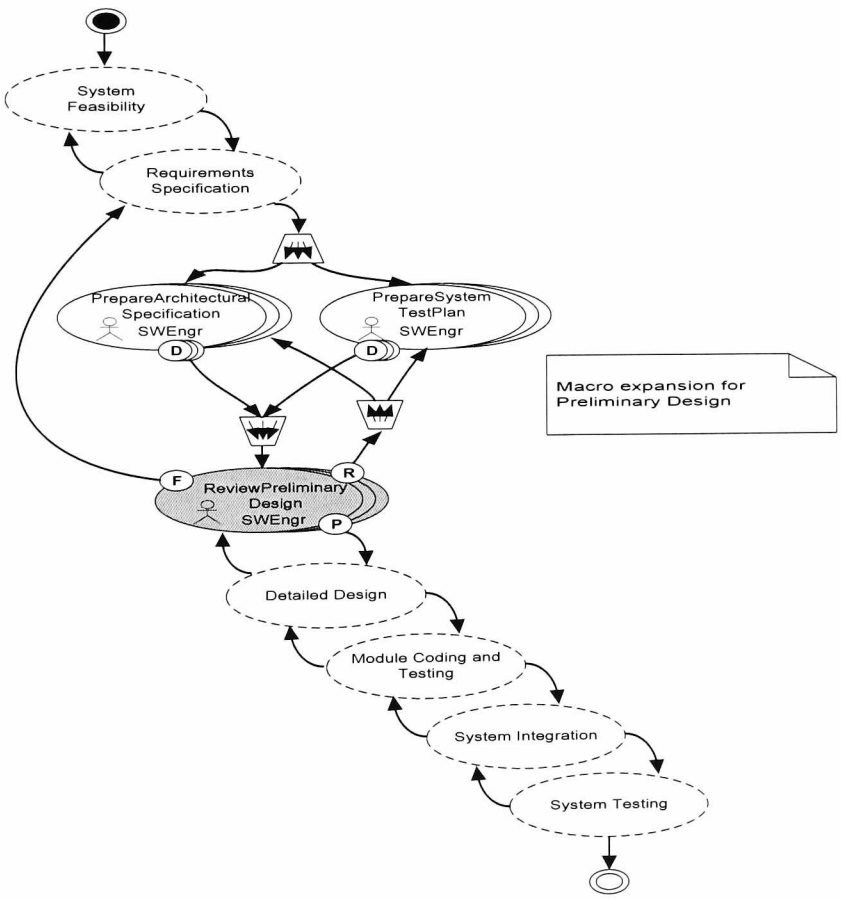


Figure B-6 Macro Expansion for Preliminary Design

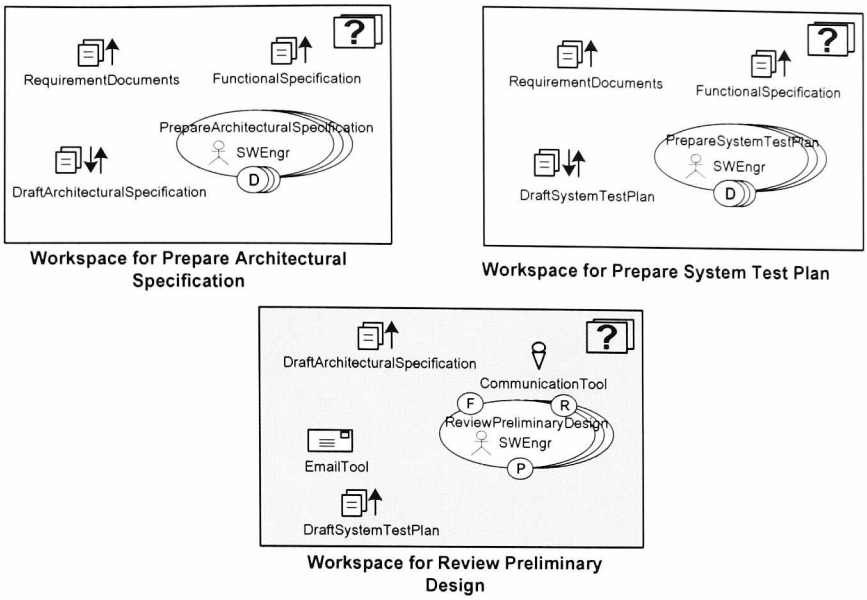


Figure B-7 Workspaces for Prepare Architectural Specification, Prepare System Test Plan, and Review Preliminary Design

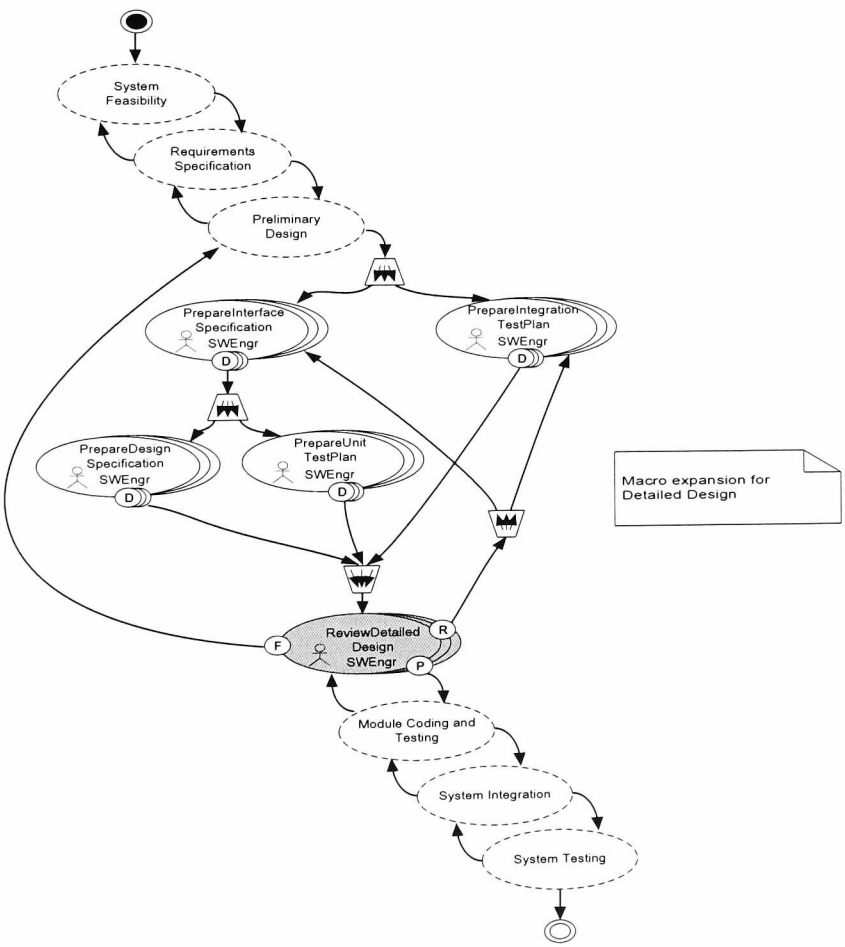


Figure B-8 Macro Expansion for Detailed Design

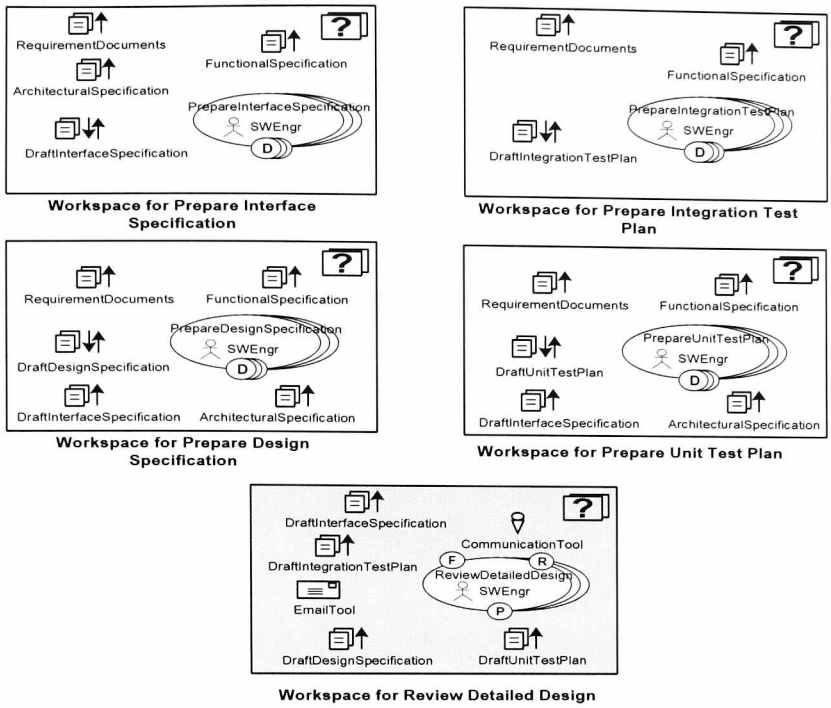


Figure B-9 Workspaces for Prepare Interface Specification, Prepare Integration Test Plan, Prepare Design Specification, Prepare Unit Test Plan, and Review Detailed Design

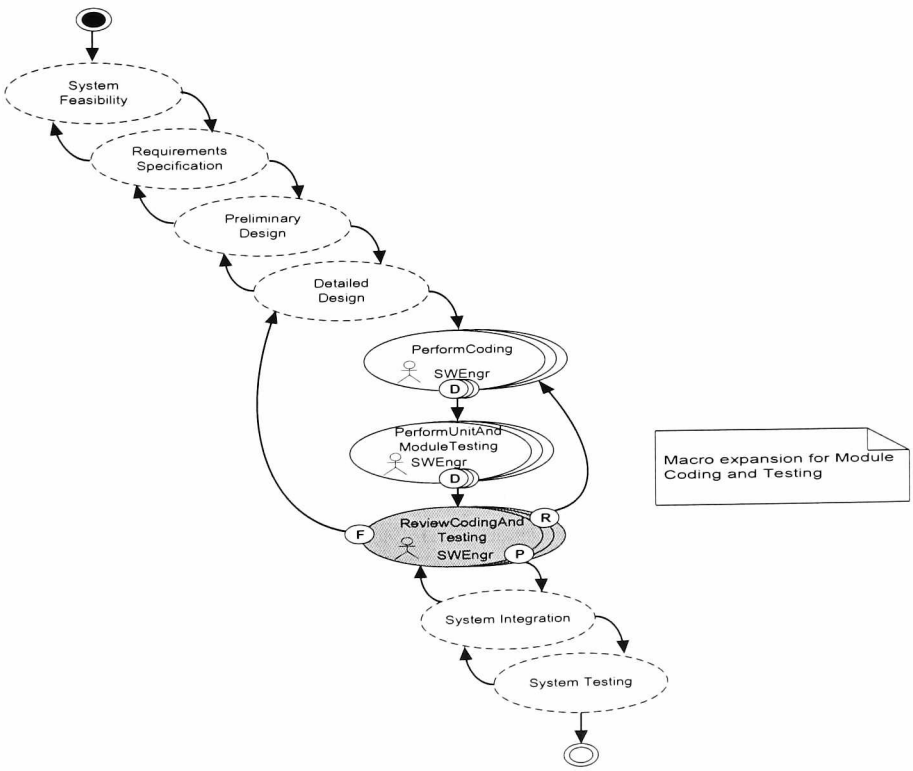


Figure B-10 Macro Expansion for Module Coding and Testing

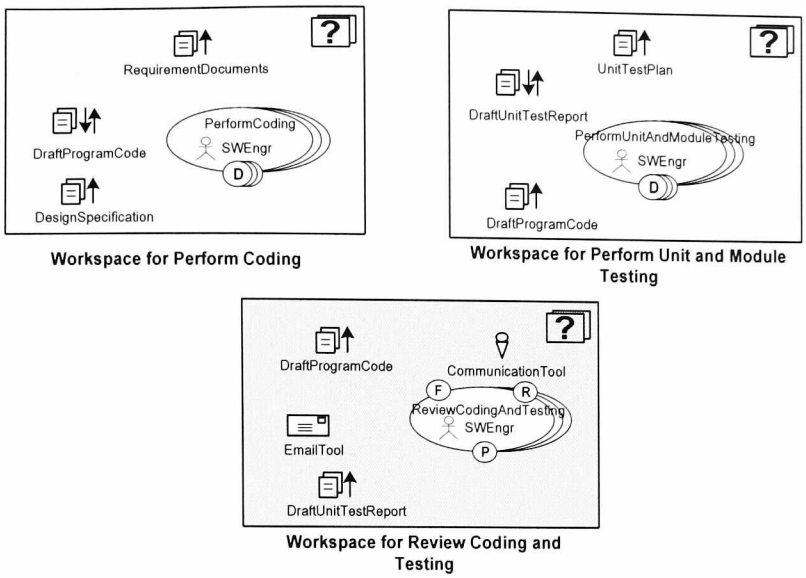


Figure B-11 Workspaces for Perform Coding, Perform Unit and Module Testing, and Review Coding and Testing

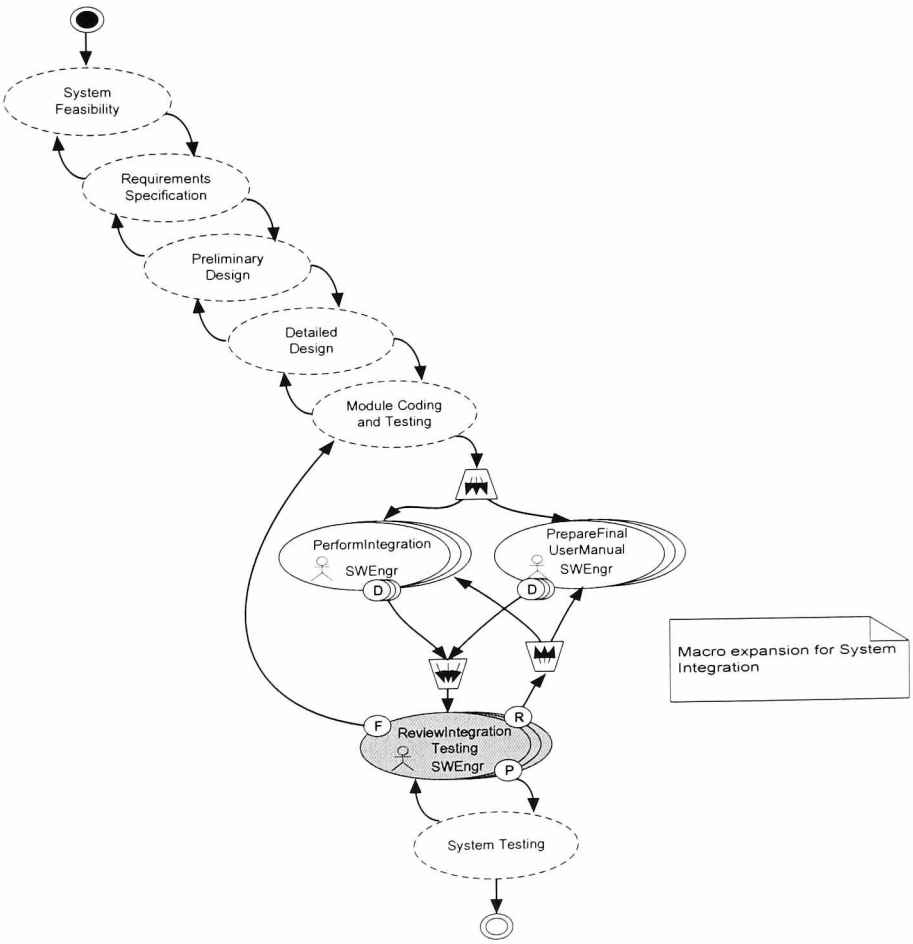


Figure B-12 Macro Expansion for System Integration

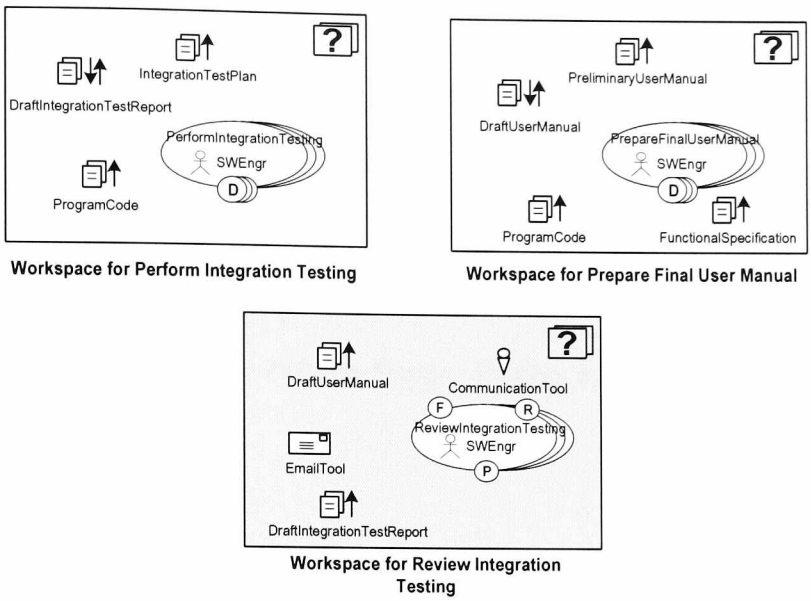


Figure B-13 Workspaces for Perform Integration Testing, Prepare Final User Manual, and Review Integration Testing

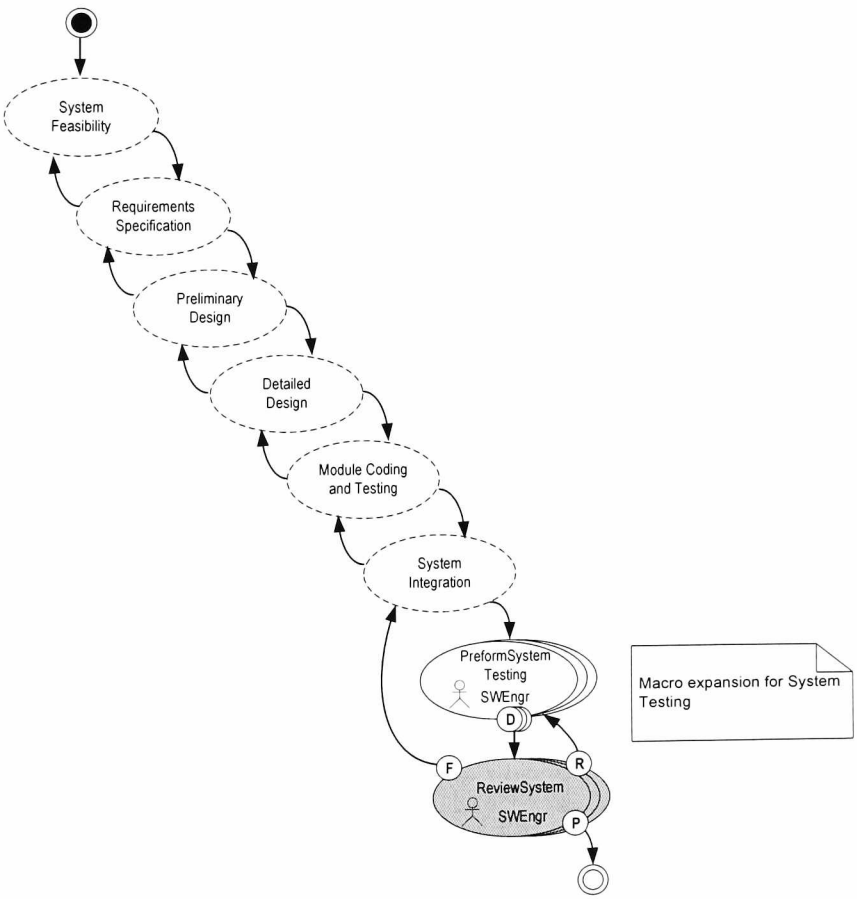
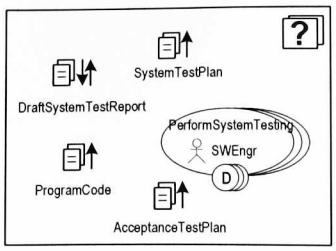
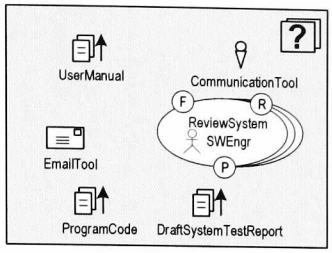


Figure B-14 Macro Expansion for System Testing



Workspace for Perform System Testing



Workspace for Review System

B-15 Workspace for Perform System Testing, and Review System