UNIVERSITY OF SOUTHAMPTON

Process Modelling for Requirements Capture

by

Stephen Crouch

A thesis submitted for the degree of
Doctor of Philosophy

Declarative Systems and Software Engineering
Electronics and Computer Science
Faculty of Engineering and Applied Science

November 2000
UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE

ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

PROCESS MODELLING FOR REQUIREMENTS CAPTURE

by Stephen Crouch

As software complexity increases, well defined managerial methods of organising software production become increasingly crucial to the success of software projects. As the investigation into the field of software process methodology continues, two approaches to process modelling have emerged as tools with which to model such processes. Graphical notations provide a clear, intuitive method of describing processes, whilst process modelling languages offer the ability to execute those processes. However, many existing process modelling languages are derived from programming languages, and, as such, inherit the low-level, syntactically complex attributes of these languages.

The main goal of this thesis is to provide a process modelling language based on a well-known and practised graphical approach to modelling processes, the Role Activity Diagram (RAD). To ensure a high level of applicability, we emphasise the importance of ensuring that the language is readily accessible to those without a technical background. To this end, as well as being a linguistic approach with origins firmly in the process modelling field, models defined in this language are intended to exhibit a high level of abstraction and intuitiveness. This thesis proposes the Romula process modelling language as a complementary approach to Role Activity Diagrams, and describes a tool developed for animating processes modelled in this language. A discussion is presented which highlights the problems of animating Romula models derived from RADs which had to be overcome when developing Romula.

Evaluation of the Romula language was achieved by validation and specification. For validation, the Romula approach was used to represent and execute two complex example process models. The first of these, named ProcMod, is a process model of a hypothetical software development process, itself validated by enacting it on a simple software development scenario. The second example process model is an implementation in Romula of a requirements-oriented process framework, presented as a method for representing the organisation of large software projects across multiple contributing entities. Examples of how this framework can be utilised are implemented in Romula. A method of translating Romula models into the formal specification notation CSP is also provided, demonstrated by example translations. This method enables Romula-derived CSP models to be checked for formal properties such as deadlock, livelock, and reachability, as well as providing a formal specification of the core semantics of the language.
Acknowledgements

First and foremost I would like to thank my supervisor, Peter Henderson, who has provided invaluable advice and constructive criticism on the production of this thesis. I would also like to thank Bob Walters, who has, on many occasions, imparted his knowledge and advice on many process modelling matters. I also must thank the other members of my research group, not only for providing astute criticism when the seminar demanded it, but for all who have helped to make DSSE the friendly and workable environment it is. I am also indebted to Margaret Crouch, my mother, for adopting the role of chief proof-reader in the later stages of this thesis. Finally, I would like to offer my thanks to Donald Cruickshank, a fellow postgraduate, without whose insight into visual programming I would have missed many a stimulating discussion.

S. Crouch
# Contents

Abstract

Acknowledgements

1 Introduction
   1.1 Background
   1.2 Scope of this Thesis
   1.3 Outline of Thesis

2 The Software Development Process
   2.1 Why Study the Software Process?
   2.2 Uncertainty in the Development Process
   2.3 Basic Software Process Models
      2.3.1 The Classic Life-Cycle and V-Shape Model
      2.3.2 Prototyping
      2.3.3 The Spiral Model
   2.4 Characterising and Standardising the Software Development Process
   2.5 Summary

3 Capturing and Enacting the Software Development Process
   3.1 Classifying Process Modelling Approaches
   3.2 Graphical Representation of Process Models
      3.2.1 The Role Activity Diagram
      3.2.2 The Unified Modelling Language
         3.2.2.1 Application of the UML
         3.2.2.2 The UML Sequence Diagram
   3.3 Linguistic Representation of Process Models
      3.3.1 Formality in the Software Development Process
      3.3.2 Classification of Process Modelling Languages
      3.3.3 Interact
      3.3.4 CSPL
      3.3.5 PML
      3.3.6 APPL/A
   3.4 Process Modelling: A One-Lane Highway?
      3.4.1 Reasoning about Process Deviation
      3.4.2 Possible Consequences of Process Deviation
      3.4.3 A Methodology is Required


### 7.3.3.2 Possible Methods of Process Improvement for ProcMod

### 7.4 Summary

#### 8 Validating Romula by Implementing a Requirements-Oriented Process Modelling Framework

- **8.1 Motivations**
- **8.2 The AMMROF Framework**
- **8.3 An Example of the Framework**
- **8.4 Encoding Example Frameworks into Romula**
  - **8.4.1 A Simple Hierarchical Process Definition**
  - **8.4.2 Encoding the Simple Hierarchical Process Definition in Romula**
  - **8.4.3 A Realistic Hierarchical Process Definition**
  - **8.4.4 Encoding the Realistic Hierarchical Process Definition in Romula**
    - **8.4.4.1 Encoding the ManageParentRelations Task**
    - **8.4.4.2 Encoding the SubRequirementListInstantiation Task**
    - **8.4.4.3 Encoding the SubRequirementListSatisfaction Task**
    - **8.4.4.4 Encoding the ManageSubContractors Task**
- **8.5 Summary**

#### 9 Conclusions

- **9.1 Contributions**
  - **9.1.1 Process Modelling and Requirements Engineering**
  - **9.1.2 Tools**
  - **9.1.3 Romula and Software Engineering**
- **9.2 Evaluation of the Romula Modelling Approach**
  - **9.2.1 Fulfilment of Primary Goals**
  - **9.2.2 Fulfilment of Secondary Goals**
- **9.3 Future Work**
  - **9.3.1 Improvement of Romula**
  - **9.3.2 Further Research**
- **9.4 Closing Remarks**

#### Appendix A - Romula's Grammatical Syntax

- **A.1 Low-level Grammar Constructs**
List of Figures

1.1 Flow diagram depicting thesis structure 7

2.1 A process-oriented view of the Classic Life-Cycle or Waterfall Model 11
2.2 Basic material-oriented view of the Classic Life-Cycle Model 13
2.3 Process-oriented view of the V-Shape Model also depicting production and usage of the Validation Test Criteria and the System Test Plan 14
2.4 The Software Prototyping Paradigm 15
2.5 The Spiral Model Paradigm 16
2.6 The five levels of the Capability Maturity Model (CMM) 17

3.1 An example Role Activity Diagram (RAD) 28
3.2 A role with two separate threads 29
3.3 The UML Outline Development Process example 30
3.4 The UML sequence diagram for a use case for order processing 32
3.5 Example process program given by Osterweil in his paper Software Processes are Software Too 34

3.6 Example of an Interact activity 38
3.7 A CSPL definition of a modify design activity 39
3.8 PML segment which forms part of an implementation of the ISPW-6 process example 41
3.9 Part of an example APPL/A definition 43
3.10 Levels of representation between the real world and the linguistic model 48

4.1 An example of how concurrency is demonstrated using the RAD notation 52
4.2 How Figure 4.1 is expressed in Romula 53
4.3 Simple Concurrent example 54
4.4 Representation of Figure 4.3 to demonstrate continuance 54
4.5 Example of interaction in a RAD 54
4.6 Representation of RAD interaction in Romula 55
4.7 Example of RAD conditional statement 55
4.8 Loop example using ExpectJoin and Join 56
4.9 Loop example using ExpectLink and Link 56
4.10 Simplified RAD example using external events 57
4.11 How external events are expressed in Romula 57
4.12 Example of role instantiation in Romula 57
4.13 Simple example of communicating roles 58
4.14 Romula representation of shop example given in Figure 4.13
4.15 Romula Representation of Figure 4.13 which deals with instances
4.16 Romula representation of a TaskForceManager and TaskForce hierarchy
4.17 Example of an allocatory role, assigning barbers to customers and vice versa
4.18 Overview of how Romula's User Interface Handler, its Interpreter (Thread Control Handler and Language Parser), and the operating system's windowing interface fit together
4.19 Layout of the animator's message window
4.20 Example of a role window
4.21 Example of a concurrent window
4.22 Example of an event window
4.23 Example of using the Materials abstraction
4.24 RAD representation of the EPOS model
4.25 Romula representation of the EPOS nameserver
4.26 Romula representation of the EPOS PLU
4.27 Romula representation of the EPOS Till
4.28 Example of a State Transition Diagram (STD) taken from Pressman
4.29 Romula representation of a State Transition Diagram
4.30 Executing the Romula representation of the State Transition Diagram with MonitorSystemStatus waiting for an incoming event
4.31 Executing the Romula representation of the State Transition Diagram depicting MonitorSystemStatus having received an incoming event fired from ReadUserInput
4.32 An example of a declarative goal-based style of process modelling
4.33 The relationship between a PHASE and its ROLES
4.34 A data flow representation of the artifacts involved in the integration phase of the pseudo-realistic process model ProcMod
4.35 Romula's constructive representation of the Integrated System artifact

5.1 Example RolEnact definition of a barber from a version of the barber shop model
5.2 Illustration of how multiple role instances communicate in mutual (Interact) interactions
5.3 Illustration of how multiple role instances communicate in hierarchical submissive interactions, where RoleA is the dominant role, and the others are submissive
5.4 Simplified barber shop example
5.5 Romula representation of the simplified barber shop example given
5.6 Romula representation of example in Figure 5.4 which uses instance lists

5.7 CSP specification of the simple Romula barber shop model given in Figure 5.6

5.8 Alternative barber shop model which introduces a simplified Receptionist

5.9 Romula representation of the barber shop model given in Figure 5.8 which has a Receptionist introduced

5.10 CSP specification of the Romula barber shop model given in Figure 5.9

5.11 The alternative barber shop model given in Figure 5.8 with an added 'checking-in' feature

5.12 Romula representation of the revised barber shop model given in Figure 5.11

5.13 CSP specification of the revised Romula barber shop model given in Figure 5.12

5.14 Introducing concurrency into the barber shop model

5.15 Romula representation of the barber shop model in Figure 5.12 which has concurrency

5.16 CSP specification of the more complex Romula barber shop model given in Figure 5.15

6.1 Shifts in project phase emphasis during the development process

6.2 Example structure of a requirements hierarchy

6.3 Example of a requirements list representation

6.4 A requirements satisfaction model based on a requirements list hierarchy

6.5 CASE Product Types

6.6 Three-level representation of an example information system in the CoNeX prototype

6.7 Partial screen-shot of zooming into the design history using conceptual descriptions and dependencies using the CoNeX prototype

6.8 The CoNeX prototype control panels

6.9 ASPEN Information Models

6.10 ASPEN Project Activity Model

6.11 The ASPEN Tool Architecture

7.1 Fragment of ProcMod, involving Requirements Analyst and Customer roles

7.2 Corresponding Romula Customer role segment from Figure 7.1, depicting how this role handles an interaction from another role

7.3 Corresponding Romula Requirements Analyst segment from Figure 7.1,
depicting usage of concurrency and interaction

7.4 RAD example of a badly structured model 132
7.5 Improvement on the RAD in Figure 7.4 132
7.6 Portion of the Project Manager role, depicting a possible re-engineering of the design phase 135
7.7 Portion of the Project Manager role in Romula, depicting how design re-engineering is represented 135

8.1 Example instantiation of the AMMROF framework 144
8.2 Simple contractor example as a Role Activity Diagram 147
8.3 Romula representation of the simple contractor example given in Figure 8.2 149
8.4 Role Activity Diagram of the realistic contractor example 152
8.5 Partial Romula representation of the realistic contractor example given in Figure 8.4 154
8.6 The problem of representing a request interaction in RADs without asynchronous events 155
8.7 Concurrent definition of the ManageParentRelations task 156
8.8 Concurrent definition of the RequirementSubListInstantiation task 157
8.9 Concurrent definition of the SubRequirementListSatisfaction task 158
8.10 Concurrent definition of the ManageSubContractors task 159
8.11 Representing part of the ManageParentRelations task as an If statement in a condition 160

B.1 Requirements Analysis Phase of ProcMod v5.2 175
B.2 Design Phase of ProcMod v5.2 176
B.3 Component Engineering Phase of ProcMod v5.2 177
B.4 Integration Phase of ProcMod v5.2 178

C.1 Requirements Analysis Phase of ProcMod v6.0 186
C.2 Design Phase of ProcMod v6.0 187
C.3 Component Engineering Phase of ProcMod v6.0 188
C.4 Integration Phase of ProcMod v6.0 189

D.1 Requirements Phase DFD 190
D.2 Design Phase DFD 190
D.3 Component Engineering Phase DFD 190
D.4 Integration Phase DFD 190
1 Introduction

It had increasingly become apparent during the 1980's that existing methods of conducting an organisation's software development process were inadequately dealing with the construction of more and more complex software systems. More specifically, the ad hoc development control implied by the manager/programming team ethic was beginning to become outmoded and incapable of maintaining a consistent method of implementing software. As the demand for feature-rich software increased in number and complexity, so did the demand for quality in software. A more disciplined approach was required beyond the simple ad hoc adherence to the traditional model of the software lifecycle which had been ubiquitous since the 1970's. Thus, the quality of software and the quality of the methods utilised to produce that software came under scrutiny, and two areas of research emerged in response to these concerns: software process modelling (the definition and analysis of an organisation's development process), and requirements engineering (well-defined, improved methods of gathering software requirements).

1.1 Background

As the software process itself increasingly became the subject of examination, the research area of software process modelling began to surface during the mid to late 1980's. Methods of modelling such processes fell into two distinct categories: textually-based description, or graphically-based description. The commonly used phrases used to describe these approaches were process programs (or process modelling languages) and process modelling notations respectively. Yet it should be noted that both of these perspectives can be described as languages, in terms of syntax and semantics. However, 'language' is a term more commonly associated with process programs.

During 1986 Praxis developed a method for describing software development processes which later evolved into the Role Activity Diagram (RAD) notation [Ould 1995], a business (not software) -oriented method for describing organisational processes. Although this popular notation originated from modelling the development of software, it was extrapolated to encompass the superset, as it were, of all types of process: the business process. Another important notation that can be used to model organisational process that has gained popularity in recent years is the UML sequence diagram [Rumbaugh et al. 1999].

The notion of a process program was introduced by Osterweil in his key paper published in 1987 [Osterweil 1987]. He emphasised the nature of a process, that it is 'a systematic approach to the creation of a product or the accomplishment of some task'. By exploiting the systematic attribute of processes, he
contended that processes can be described within the context of a software programming-like language. He describes these process programs as 'namely an object which has been created by a development process, and which is itself a software process description'. The term process modelling language thus came into use, and many of these began to appear (see chapter 3). A distinct property of modelling processes as programs is that they could be executed on a computer, with the benefit that they could be enacted; which enables an executing process program to interact with its inherent real-life process which it models. Thus, it could be possible for an enacting process program to adopt a suggestive, or even controlling, role in the process it represents. In addition, the possibility exists that since the process is stored on computer and can be executed, tools could be built which analyse the process and its effectiveness possibly during enaction, and possibly in real-time.

Formal definition of a software process, either by process program or notation, clearly has its advantages. It is concise, it is explainable, and in some respects it provides accountability: if a process fails for whatever reason, by examining the process, it is possible to ascertain the cause of the failure, and improve the process accordingly. Yet, more important than these reasons is the fact that a process definition is exactly that: a defined process. By establishing an accessible definition of the process, an organisation has moved beyond the ad hoc method of performing tasks, and is aware at any point in the development process what tasks are being performed to engineer a piece of software, and who is involved in performing those tasks. This definition of an organisation's process helps those involved in the process to understand their part in the larger whole, and to appreciate the reasons behind their activities [Ould 1995]. As an example of a tangible benefit of process definition, such managerial advantage allows for better time (and therefore cost) estimation with respect to software projects. They have become predictable. However, despite the advances that have been made in recent years, process modelling is still comparatively a new area of interest, and process modelling languages and process modelling notations have their weaknesses.

Many process modelling languages appear to have a distinct problem: they attempt to describe process modelling in terms of programming concepts and constructs. Although this approach can be used to describe process models, is it appropriate? This approach requires an obvious knowledge of these programming language techniques, and more importantly, how to apply them to the fundamentally different area of process modelling. Essentially, who will be responsible for modelling a process? It could logically be assumed that those responsible would have relevant knowledge of process modelling concepts and techniques. It should not be assumed that they have (or wish to have) any programming expertise. Process modelling is not itself a software-based discipline: it is a discipline which models any organisation's process, not only a software organisation's process. Just because we are modelling a software process it does not necessarily follow that we should use software engineering concepts to model that process. This leads to another disadvantage. By describing a process model using programming language constructs, the model is also not as understandable to others without this knowledge. What is the point of a process model if only the programmers can create it and understand it? For these reasons, would it not be more logical to
develop software process models with *modelling*-based syntax and semantics? The syntax and semantics of notational modelling approaches embody modelling concepts which can more easily describe process models. However, notation modelling approaches also have a drawback. They cannot easily take advantage of the possible analytical abilities offered by a computer, unlike process modelling language approaches.

A method which overcomes the shortfalls of both of these approaches would allow processes to be modelled in terms of modelling concepts, and hence be easier to understand within that context, as well as being readily executable and (therefore) enactable. This thesis will present a process modelling language and associated tool that attempts to fulfil these requirements.

At around the same time as software process modelling began to appear, the area of requirements analysis was also in its infancy. It was a development phase that was not awarded appropriate effort or respect up until the mid 1980's, since although it was essential to determine what the requirements of the project were, it did not contribute anything of any actual tangibility to a software project. The later phases of design and implementation were deemed more important, since they inevitably led to the construction of the actual software system.

However, as the complexity of software systems increased, development of those systems demanded greater attention to a more complex set of requirements. In addition, as organisations became more dependent on the software systems that supported their business, software quality became more of an issue. Gradually, requirements analysis proponents such as Chung, Mylopoulos and Nixon [Chung 1991a, Chung et al. 1991b, Mylopoulos et al. 1990] began to introduce the idea that requirements can be categorised into two types: functional and non-functional. Functional requirements dealt with the functionality of the desired system, whilst non-functional requirements dealt primarily with quality concerns. The authors’ concern was that non-functional requirements ‘as far as software engineering practice is concerned, ... are generally stated only informally during requirements analysis, are often contradictory, difficult to enforce during software development and to validate for the user once the final system has been built’ [Mylopoulos 1992]. Understanding the conceptual separation between these two types of requirements is essential in producing a complete and accurate Requirements Specification for a software project.

The importance of requirements is also highlighted by the DAIDA approach [Jarke 1991a, Jarke 1991b] and the ASPEN framework [Doheny and Filby 1996]. Each of these define various models which describe software, identifying requirements as an integral and significant part of software and the development of software. DAIDA, for example, only allows design decisions to be made in relation to the requirements model, thus ensuring that a complete set of requirements has been defined.
1.2 Scope of this Thesis

The problems associated with current process modelling approaches affect their applicability. Process modelling languages are complicated and have their roots firmly within their origins of programming languages, and for this reason they are less easily usable or understood by those without programming expertise. However, due to their formal, textual nature, they can be analysed and executed by software. Notational approaches cannot as easily be exploited by computer software, yet provide an easier and more appropriate interface for constructing and understanding a process model.

The main goals of this work are as follows:

- **[PRIMARY]** To develop a textual method of process modelling which embodies the understandability of notational methods with the exploitative power of the process modelling language.
- **[PRIMARY]** To develop an associated animation tool that allows models developed in this modelling language to be executed.
- **[PRIMARY]** To validate how well this modelling approach could model and enact a pseudo-realistic process model. By pseudo-realistic we mean a model which is realistic in size and scope, and thus adequate for validation purposes, but is not actually a real process.
- **[SECONDARY]** To validate the realism of the pseudo-realistic process model by enactment using the modelling language and animation tool to ascertain if the validation process is itself valid, and to validate further the modelling language's applicability for process modelling.
- **[SECONDARY]** To validate further the modelling language by using it to create a model which offers process support for the handling of software requirements in a requirements-oriented context.

This work follows on from existing process modelling research undertaken within our research group. RolEnact [Phalp et al. 1998] is an event-driven process modelling language that is used to model and execute business process models, and is discussed in chapter 5.
1.3 Outline of this Thesis

The outline of the thesis is as follows:

- **Chapter 2: The Software Development Process**
  This chapter introduces the software development process, and why it should be studied. Uncertainty within the development process is examined, and some basic process models that are in use today are introduced as examples of modelling the software development process. Popular methods of evaluating an organisation's software process are explained.

- **Chapter 3: Capturing and Enacting the Software Development Process**
  This chapter deals with how the software development process can be captured (defined), and enacted. A classification scheme for process modelling approaches is presented. Two popular graphical notation-based approaches (RADs and UML sequence diagrams) are discussed, as are various linguistic (textual) methods of process modelling. Formality in the software development process is discussed, and a classification scheme for process modelling languages is introduced. The issue of deviation from a process model is also discussed.

- **Chapter 4: Introducing Romula**
  This chapter describes the process modelling language, Romula, and its animation tool which have been developed within the department. The method used to verify the Romula language is explained, and a possible extension to the Romula language definition which exploits the nature of the software process is described. A discussion is given of Romula in comparison with other process modelling languages, and three unexpected applications for Romula are presented.

- **Chapter 5: Formally Specifying Romula in CSP**
  In this chapter the core semantics of Romula are discussed and specified using a formal notation, CSP (Communicating Sequential Processes). This allows us to reason about Romula models formally, and understand what a Romula model actually means in formal terms.
Chapter 6: The Software Process: Requirements in Context

This chapter introduces and details some issues related to requirements analysis for the purposes of background for the next two chapters. The nature of both functional and non-functional requirements are explored, and two CASE tools which manage the software development process are introduced which emphasise the importance of requirements.

Chapter 7: Validating Romula with a Pseudo-Realistic Process Model

This chapter presents the method and results of validating the Romula language and animator. It introduces a pseudo-realistic process model, ProcMod, which was used for validation. In addition, a critical examination of ProcMod itself is detailed, to validate further that Romula is a viable process modelling approach.

Chapter 8: Validating Romula by Implementing a Requirements-Oriented Process Modelling Framework

This chapter introduces a process modelling framework which allows requirements and their associated development process to be represented within the context of a hierarchical system of contracting companies. Although it demonstrates this is a complex task, the feasibility of this approach was additional evidence that Romula is a viable process modelling approach.

Chapter 9: Conclusions

This final chapter summarises and evaluates the contribution of Romula to process modelling, placing it in context with other process modelling approaches. This chapter also discusses the issues raised during Romula's development.

Appendices: Appendix A provides an overview of Romula's grammatical syntax. Appendices B and C present contiguous versions of a pseudo-realistic process model, the first of which was used to validate the process modelling language developed. Appendices D and E illustrate a DFD and Romula representation respectively of a method detailed in section 4.8.3 for describing a complex process model in terms of partial ordering. Appendix F contains a Romula representation of an example requirements-oriented framework which is described in chapter 8.

The structure of the thesis is illustrated in Figure 1.1. Each arrow denotes the conceptual relationship between corresponding chapters.
Figure 1.1: Flow diagram depicting thesis structure

The Research level indicates those chapters which originated from research conducted during the course of the project. The Implementation level indicates the chapter where research was culminated from the Research level into a prototype to demonstrate proof of concept. The Validation level denotes those chapters which lend evidence of validity to the approach undertaken. The Conclusions level brings together the results and provides a forum for discussion on the contributions made to software engineering by the chosen approach.
In this chapter we will introduce the overall context of this thesis: the software process. Section 2.1 will discuss the motivations for studying the software process, and section 2.2 will elaborate on the inherent uncertainties within this process which increase project risk. Section 2.3 will introduce and discuss three basic software process models, and section 2.4 will discuss how the software process may be characterised and standardised.

2.1 Why Study the Software Process?

There is a large body of evidence ([Humphrey 1988, Krasner et al. 1992, Lehman 1991, Osterweil 1987] in particular) which stresses the need to define a disciplined development process and in addition to develop the necessary support technology. Many predefined process models exist and are in use, and some of these will be described in section 2.3. Defining a development process model increases overall team understanding of project responsibilities and commitments, thus reducing project risk. However, it is not simply enough to define such a model. In large scale projects, adhering to the detailed process may be critical to project success [Lehman 1991].

In an effort to provide a summary organising all aspects of software engineering into an overall context, Lehman [Lehman 1991] focuses on generalising the basic concepts of software engineering, providing ‘a unifying framework, a basis for conceptual advance’. He categorises three types of system development uncertainty, covered in section 2.2, which may substantially contribute to project risk.

2.2 Uncertainty in the Development Process

Uncertainty in developing software represents project risk. Lehman identifies three specific types of uncertainty associated with system development:

- **Heisenberg-like uncertainty**: As a system progresses through its lifecycle, more needs (requirements) are discovered which were not initially identified; the system therefore constantly evolves, sometimes very quickly. Clearly, new and old
requirements need to be handled proficiently, and related, so a consistent and unconflicting new system is developed.

- **Embedded uncertainty**: Throughout the modelling process, assumptions (implicit or explicit) concerning an application, its operational domain and the solution system are inevitably introduced which will become invalid. Therefore, they must be updated and kept valid by ‘appropriate changes to individual elements and/or to their documentation’.

- **Process uncertainty**: Similar to embedded uncertainty, assumptions can be made in relation to the systems development process and software: 'However convincing the demonstration that a program is correct or satisfactory, the possibility remains that an erroneous argument is being used in the [software] refinement process'.

However, it is *Heisenberg-like uncertainty* and *Process uncertainty* which are most relevant to this thesis. A process modelling language is introduced in chapter 4 which aims to decrease the process uncertainty related to an organisation's software development process. Chapter 8 introduces a framework which aims to reduce Heisenberg-like and process uncertainty between sub-contracting organisations by proposing a method which allows requirements and their associated development process to be represented together.

Reducing Heisenberg-like uncertainty in the software development process requires catering for *requirements change*. This is achieved by defining procedures which allow a requirements framework (see section 6.3) to be updated whilst ensuring it is still consistent and contains no conflicts. For example, adding a requirement that describes a fast user input rate for an air traffic control system may contradict an earlier requirement for a certain type of user interface. A strict procedure for adding requirements would be necessary to recognise and express these types of relationship.

To reduce process uncertainty requires methods for *software process management*. This involves defining and following software development processes which can ensure certain software engineering standards are adhered to. For example, a software process which either incorporates the ISO 9000 series of standards, or achieves a certain level of process maturity (see section 2.4). Inclusion of such engineering practices can yield tangibly better quality in software.

Embedded uncertainty, however, is not explicitly addressed within this thesis. Such uncertainty arises from assumptions within the definitions and models of the real world; the operational domain. Such abstract views are dependent on the methods and model representation employed to create them, and it is specifically issues with the software process and the requirements engineering process which are addressed by this thesis.
Lehman [Lehman 1991] makes a salient point concerning an uncertainty principle of computer application:

*The outcome, in the real world, of software system operation is inherently uncertain with the precise area of uncertainty also not knowable.*

Therefore, it is by recognising uncertainty as an inherent risk in developing software that we can fully appreciate the pitfalls as well as the promise of developing software.

The software giant Microsoft owes much of its success to its development method. Their development process, called 'Synch-and-Stabalize' [Cusumano and Selby 1997], is an iterative form of Rapid Application Development. Essentially, a project is split into modular pieces, and each piece is developed in conjunction with all other teams. However, by ensuring that the *overall* program can be recompiled at the end, or the beginning, of each day, errors can be located and corrected on a day-to-day basis. This method also ensures that they 'always have a product ready to ship', which is Microsoft's core development mentality. Thus, the risk inherent in Heisenburg-like uncertainty is significantly reduced.

A far more haphazard, yet surprisingly effective, development strategy is that of Open Source [Open Source 2000a, Raymond 2000, O'Reilly 1999]. The Open Source development paradigm is responsible for and commonly associated with the Linux phenomenon [Linux 2000]. Essentially, when software is available freely on the internet for reading, modification, and redistribution, it evolves, and often at an alarming rate. It makes use of the collective IQ of many programmers on the internet by providing the promise of *intellectual challenge*. By introducing and publishing problems on the internet, individuals come forward and implement solutions to these problems which, when taken as a whole, contribute towards a greater product. The operating system Linux is a product of this development mentality, and proof of its effectiveness. Increasingly over recent years the Linux operating system has emerged as (at least) a viable contender in the operating system world where the ubiquitous Microsoft Windows operating system has enjoyed a dominant market share. If further proof were needed, Open Source development is responsible for the tremendously popular Apache web server, and also the fundamental BIND and sendmail internet software [Open Source 2000b], which provide the backbone for the entire internet.

### 2.3 Basic Software Process Models

This section introduces four of the more well-known general process models used as a basis for the construction of software in industry today:

- The Classic Life-Cycle.
The V-Shape model.

Prototyping.

The Spiral Model.

These models represent a cross-section of the current state of software construction, and by examining them we can ascertain the needs of the modern software organisation and the approaches they employ to produce software.

2.3.1 The Classic Life-Cycle and V-Shape Model

The classic life-cycle, or waterfall model, as shown in Figure 2.1 is the oldest and still the most widely used paradigm for software engineering [Pressman 1994]. Although some improvements have been made to this model [Lott 1997], the classic version of this model will be discussed here since it is the version most commonly in use.

![Figure 2.1: A process-oriented view of the Classic Life-Cycle or Waterfall Model](image)

This conventional lifecycle paradigm encompasses the following activities. The description of these activities, some or all of which are common to many process models, provides an overall context for the research presented in this thesis.

- **Requirements Analysis**: Various analytical activities investigate and model the *information domain* for the software, providing an operational framework within which the software will operate. It is the determination of the customer's software requirements which remains the most substantial and important activity, however. Key success in this area demands a well-structured and disciplined approach to requirements gathering, and close correspondence with the customer.
Numerous methods exist to represent and aid the understanding of the requirements, such as Telos [Mylopoulos 1990], which allows the requirements analyst to model both the software's operational domain and the software's functionality within that operational domain. A well-substantiated description of requirements can form an integral part of assessing the validity of the completed software. A series of validation test criteria are created which are used during validation testing to ensure that the overall software product satisfactorily meets requirements.

- **Design:** Requirements are translated into a representation of the software which can be assessed for quality, by customer and/or internal technical reviews. Translation comprises four elements: data structure, software architecture, procedural detail, and interface characterisation. With the advent, proliferation and acceptance in recent years of the graphical user interface (GUI), greatly increased focus is upon providing a well-designed, quality interface for the user. The central goal of design is to produce a representation which can be mechanistically translated into a consistent machine-executable format. After software architecture and procedural detail have been completed, a test plan is devised which is used during the testing stage to ensure that the software's modules produce expected results from test input.

- **Coding:** Design is translated into a machine-executable format. Assuming the preceding design is sufficiently detailed, this process can be accomplished mechanically.

- **Testing:** This comprises two activities. Firstly, the software is tested with respect to a test plan produced during the design stage. This activity verifies that the internals and functional externals of the software are performing logically and as expected. Secondly, the software is validated with respect to validation test criteria established during requirements analysis. This determines whether the software conforms to the defined software requirements.

As the requirements phase of development has increasingly become subject to scrutiny [Chung et al. 1995], and new methods of requirements analysis and representation are being proposed, the task of providing quality software validation has increased in importance. Software contracts can often stipulate successful software validation as payment criteria. Additionally, as software systems become more complex and subsequent development times increase, it follows that the operational domain itself may well have evolved during this period. For these reasons, the importance of ascertaining the applicability and validity of the eventual software remains considerable.
• **Maintenance**: An inevitable, if unfortunate, circumstance of developing software is that it will undergo change after it is delivered to the customer. Such changes can occur for a variety of reasons: errors may have been encountered, the operational domain of the software has altered, requiring subsequent alteration to the software, or the customer requires functional or performance enhancements. However, the use of 'formal software development tools' which facilitate development "replay" of the software development phases can reduce the workload inherent in this phase [Jarke 1991].

To promote further understanding of this model, a basic, material-oriented view is given in Figure 2.2. This illustrates the production and use of materials by the different phases. Since it is the development of the software itself which is within the scope of this thesis, the *Maintenance* phase is not included.

![Diagram](image.png)

**Figure 2.2**: Basic material-oriented view of the Classic Life-Cycle Model
A conceptual extension to the waterfall model is the V-Shape model illustrated in Figure 2.3; initially developed by the UK Ministry of Defence and commonly used by ICL [ICL 1988]. It takes advantage of the document-flow depicted in Figure 2.2 to stress fundamental relationships between the Requirements Analysis and Validation activities, and the Design and System Test activities.

\[ \text{Figure 2.3: Process-oriented view of the V-Shape Model also depicting production and usage of the Validation Test Criteria and the System Test Plan} \]

Not only does Requirements Analysis and Design provide the test criteria for the Validation and System Test activities, by their intrinsic nature Validation and System Test are dependent on Requirements Analysis and Design. This V arrangement highlights the importance of this correlation, and reveals a natural balance of activities. It is also important to note that this model defines Validation as a separate activity from System Test, unlike the waterfall model.

2.3.2 Prototyping

Software prototyping [Pressman 1994] has a task sequence similar to the waterfall model, but utilises an iterative form of software development. The developer firstly creates a model of the system to be built. This model can take one of three initial forms:

1. A paper prototype, or appropriate PC-based model depicting how the Human Computer Interface (HCI) will operate.
2. A working software prototype which only contains a subset of the desired overall functionality.
3. An existing piece of software that will be extended and improved upon during development.
Figure 2.4 illustrates the prototyping model. As with the waterfall and V-Shape model, the prototyping paradigm begins with a requirements gathering phase. Similarly, customer and developer meet and discuss the initial requirements for the software. Once achieved, a brief phase of design is conducted, which essentially centres on the HCI elements required for the software which will provide access to the software's functionality. Based on this design, a prototype is constructed which forms the basis for an evaluation by the customer. This evaluation leads to refinement of the initial software requirements, and the process is then re-iterated continuously from *Quick design* to *Refining prototype* until the eventual software satisfactorily meets the customer's needs. After the prototype is complete, it may be 'thrown away', and information gained from this iterative process is then used as a solid basis for constructing the final piece of software in the *Engineer product* phase.

![Figure 2.4: The software prototyping paradigm](image)

However, despite the obvious advantage of increased customer-developer interaction during the development phase, the developer must ensure that the customer is aware of the 'throw away' nature of the actual prototype. Otherwise, the customer may misunderstand that the prototype is not the final product, but only a development 'stepping stone' to a properly engineered solution.

### 2.3.3 The Spiral Model
This spiral model is a hybrid of the waterfall and prototyping models, with an added phase: *risk analysis*. The model defines four major activities which are enacted in an iterative manner until a completed product is eventually produced:

1. **Planning**: Concerned with determination of project objectives, alternatives and constraints.

2. **Risk Analysis**: Alternatives are analysed, and risks are identified and addressed.

3. **Engineering**: Development of the 'next-level' product.

4. **Customer Evaluation**: The customer assesses the results of the previous engineering phase.

An illustration of the spiral model is given in Figure 2.5. With each iteration of the spiral, progressively more complete versions of the software are built. The first circuit around the spiral is concerned with initial requirements, objectives, and project constraints, and risks are identified and analysed. If *Risk Analysis* identifies an unacceptable degree of risk in requirements, prototyping is utilised to create simulations and various other models to clarify any problems and refine requirements (*Engineering*). The customer then evaluates the engineering work (*Customer Evaluation*), making suggestions for modifications. The spiral then begins again, with each successive turn of the spiral working towards a more completed and satisfactory product. At each loop of the spiral, a 'go, no-go' decision is made after *Risk Analysis*. If project risks are deemed to be too great, the project can be terminated.

![Figure 2.5: The spiral model paradigm](image-url)
If properly enacted, the spiral model paradigm can ensure risks are significantly reduced. Since risk analysis is an integral part of the model, usage of this paradigm ensures that development uncertainties are kept to a minimum. However, this is also a disadvantage of the model. Proper utilisation of this paradigm requires considerable risk assessment expertise, and without this, project failure becomes a far greater possibility.

2.4 Characterising and Standardising the Software Development Process

When developing process support software, an important factor to consider is the process philosophy of organisations for whom the software is targeted. Humphrey [Humphrey 1988] proposes a five-level system of software process maturity (see Figure 2.6) which forms part of the SEI's Software Process Assessment (SPA) method. It focuses on clarification and possible improvement of an organisation's actual software process.

![Figure 2.6: The five levels of the Capability Maturity Model (CMM)](image)

Proposed in this paper is a rough guideline to identify at which level a software organisation resides in the presented maturity framework:

1. **Initial**: An ad hoc development process exists, neither documented nor adhered to.
2. **Repeatable**: Via a stable, repeatable process, the organisation utilises rigorous project management of commitments, costs, schedule, and changes.
3. **Defined**: The development process has been defined; it provides a basis for better understanding of the process.
4. **Managed:** Comprehensive process measurements have been introduced, beyond those of cost and schedule performance. Improvements in process quality begin to manifest.

5. **Optimising:** A solid foundation exists for continued process improvement and optimisation.

Additionally, instructions are given which an organisation at a certain level in the framework can follow to progress to the next level of process maturity. An interesting, yet disturbing, statistic given by the Software Engineering Institute in this paper is that in 1988 only 14% of several dozen organisations queried have process maturity above level 1. Also, at this time no organisation had a maturity level of 3. In addition, those falling below this level only have an average maturity level of approximately 0.6. Recent years, however, have seen the proliferation of the SEI’s SPA into the commercial arena, and several case studies within such organisations (including Motorola, Schlumberger, Bull HN, and Siemens) have highlighted the impact of using the CMM.

Much later statistics from the SEI [SEI 1999], dated mid-August 1999, are more encouraging: over 55% of the 734 organisations assessed between 1995 and 1999 are level two or above, a tremendous rise from 14% in 1988. Even more encouraging is that nearly 23% of these organisations are at level three or above. The existence of the 1.4% of organisations that are at level five proves that reaching level five is not an unattainable or unrealistic goal. Another indicator to the success of the CMM is that from 1995 to 1999 27% of the assessments on file at the SEI were conducted at sites located outside of its home ground, the US.

Humphrey is objective; giving advantages for each progression in the framework, and also possible problems faced by each progression. He explains that a truly effective software process exhibits the following features:

- **Predictability:** The process is firstly predictable; estimates and schedule commitments are met with reasonable consistency, and the quality of the resulting products generally meet user needs.

- **Statistical Control:** If the software development process is under statistical control, improving the process is the only way of *consistently* providing better results from the process.

- **Measurement:** The basic principle behind statistical control, measurement provides a method of reasoning quantitatively about the effectiveness of a process. However,
overzealous measuring can prove counterproductive; actual degradation of the process can result.

It is clear that CASE tools can aid in providing these process features. However, Humphrey argues that at the Repeatable stage (and Initial, supposedly), new tools and methods can affect how the process is performed, and new technology, for example CASE, can possibly do more harm than good. The next section discusses the applicability of software for organisations within this framework.

A revised maturity model, CMM v1.1, was later released in 1993 [Paulk et al. 1993] which characterises the properties of an immature software organisation:

- Throughout the course of the project software processes are generally improvised by practitioners and their management. Despite a possible specification of the software process, the specification is not rigorously followed or enforced.
- The organisation is reactionary; managers are usually focused on solving immediate crises [not the observation and improvement of the process, presumably].
- Since schedules and budgets are not based on realistic estimates, they are routinely exceeded.
- Product functionality and quality are often compromised when hard deadlines are imposed.
- There is no objective basis for judging product quality, or for solving product or process problems. Product quality is therefore difficult to predict.
- When projects fall behind schedule, activities intended to enhance quality, such as reviews and testing, are often cut short or eliminated.

Thus, it is apparent that in addition to product development, the organisation's actual development process itself must also command adequate focus. The key feature responsible for each of these shortfalls is predictability; the lack of enforced process and premeditated methods to solve unexpected situations increases project risk.

One of the most influential and widespread set of software quality standards throughout Europe is the ISO 9000 series [ISO 9000-1 1994]. It consists of the following standards:

- **ISO 9000-1**: This assumes the role of 'road map for the ISO 9000 family'. It provides a consistent overview and set of author's assumptions concerning ISO 9000.
ISO 9004: This series provides guidance for quality management, for the purposes of design, implementation, and improvement of a quality system. This series is based on the assumption that all work is accomplished by a process (the software process), and, accordingly, quality management deals with managing all processes used within an organisation.

ISO 9001, 9002 and 9003: These are models used for external quality assurance, and provide a set of requirements for an organisation's software process to fulfil. Conformance to these requirements leads to certification or registration, and provides quality assurance and confidence to customers that specified quality requirements will be met.

ISO 9000-2 and ISO 9000-3: ISO 9000-2 is a guideline for the application of ISO 9001, 9002 and 9003, whilst ISO 9000-3 is a guideline for the application of ISO 9001 to the development, supply and maintenance of software.

The authors of the paper, D. Stelzer et al, strongly recommend a route towards successful certification: initially, a company should use the ISO 9004 series to design and implement a quality system. Once installed, the quality models of ISO 9001, 9002 or 9003 can be used to demonstrate an adequate quality system. Following this, ISO 9001 is usually used as a basis for certification.

Despite being the vehicle for the improvement of many software supplier's products, the ISO 9000 standards continue to receive unfavourable criticism in journals, textbooks, and conferences [Honda et al. 1997, Stelzer et al. 1997]. A notable criticism made by the paper's authors themselves is that ISO 9000 does not necessarily lead to improved productivity, cycle time and product quality. Of course, this would depend on the quality of the existing organisational structure. An organisation with a good organisational structure and mature software process are naturally going to benefit less from ISO 9000 than an organisation with an inferior organisational structure and software process.

Similar in aim to the SEI's approach, although less well known to software practitioners, is the SPICE (Software Process Improvement Capability dEtermination) project [Andres 1997, Dorling 1993, Kitson 1996]. The SPICE approach hopes to integrate and advance, not compete with, the SEI's approach, by providing a standard for process assessment:

...one of the most challenging aspects of the SPICE effort will be to incorporate the best aspects of existing successful approaches and pave the way for future improvements in software process assessment without precluding the continued use of successful approaches to software process improvement.
Kitson states that many software suppliers already feel compelled to pay attention to a number of quality thrusts, depending on the organisation. Hence, by adopting SPICE as an integrating framework of their selected approaches, their established methods may remain intact within the SPICE standard. It is designed to be supportive of the ISO 9000 series of standards, and Kitson stresses that it is not simply a software variant of ISO 9001.

Philosophical and goal similarities exist between the SEI SPA method and the SPICE effort, and it could be argued that any Integrated Process Support Environment (IPSE) should at least acknowledge these, since obviously any large-scale IPSE used by a software company would greatly influence its software process evaluation, regardless of evaluation approach. Philosophical similarities are:

- A focus of continuous process improvement.
- A public and defined assessment approach.
- An underlying reference model for assessment.
- Recognition that results from different modes of method use must be comparable and consistent.

Shared goals between the two approaches are:

- Elevate the importance of continuous process improvement in both the software supplier community and the software acquisition community.
- Provide means by which objective measures of process quality can be determined in a repeatable and valid manner.

Integral IPSE support for these, or at least minimal acknowledgement of them, is clearly feasible. For example, an IPSE which provides functionality for creating and tailoring software processes, and also provides a system for evaluating those software processes with the aim of improvement, would be a significant achievement.

However, the main difference between these two approaches is that SPICE provides a framework for a model and a method, while the SEI actually provides a conformant model (the Capability Maturity Model).

D. Stelzer et al [Stelzer et al. 1997] provide some recommendations for advancing the field of software quality concepts within ISO 9000, which, they claim, may well be helpful for advancing other software quality initiatives such as CMM and the SPICE standard. Some of the more interesting recommendations are detailed below:
1. **Focus of business success:** A determined focus should be maintained on enhancing business *success*, as opposed to simply conforming to the standards. i.e. The key objectives of any concept in software improvement should be meeting customer requirements, creating value for the customer, cutting costs, reducing time-to-market, and improving return on investment. However, an interesting possibility raised by J. Herbsleb *et al.* in the CMM paper [Herbsleb *et al.* 1997] is that as a quality management system is first introduced, the customers may actually suffer initially as a result; customers may not initially like the discipline that requirements management brings to customer interactions, and that customers suffer when attention is focused internally as Software Process Improvement (SPI) gets under way. Of course, this concern may well be applicable to other software quality management concepts including ISO 9000.

- **Ensure the purpose of the concept is clearly understood and communicated:** Most problems that occur when attempting to implement the suggestions of the ISO 9000 family are due to organisations focusing solely on ISO 9001 and 9000-3 rather than on the entire ISO 9000 family when implementing a quality system. The authors state that other software quality management concepts have suffered from similar problems. The maturity questionnaire of the Capability Maturity Model, for example, is often confused with the CMM itself.

- **Promote a constructive discussion about strengths and weaknesses of the concept:** With the emergence of many separate software quality concepts, it is inevitable there will be various conflicting views, different perspectives and approaches to software quality management. Discussion of these differing perspectives provides the opportunity for improving the quality concepts for software development, and for advancing the software quality management field in general.

- **Encourage organisations to gather and publish experience with the concept:** Such reports may help to clarify strengths and weaknesses of the concept. Currently, very few software companies have done this. It should be the role of the standardisation body to encourage organisations to publish such information.

- **Provide guidelines explaining the most common pitfalls and potential drawbacks when implementing the concept:** Many software companies have suffered losses when applying ISO 9000 because they are repeating mistakes that other companies have already made. By providing guidelines which aim to explain the potential problems with a software quality improvement concept, other organisations may avoid making unnecessary mistakes when implementing the concept.
Following the recommendations given by the authors may lead to an improvement in the overall understanding of software quality improvement concepts, such as ISO 9000, SPICE, and CMM, ensuring that organisations that wish to improve or implement a decent quality management system may do so with a minimum of potential risk. In addition, by adopting a policy of honest objectivity with such standards, by publishing the advantages and disadvantages with the possible approaches, organisations will be able to make a far more balanced decision when deciding to implement an existing quality management system. With access to such objective information and guidelines, such organisations would be able to implement a quality system with an increased level of confidence.

2.5 Summary

As software becomes more complex, so does the process utilised to create this software. It is crucial that software development organisations recognise the risks and inherent uncertainties that naturally exist as a consequence of producing software.

Various models of software production exist, catering for an ever increasing number of differing development paradigms. However, these models only describe a software process in generalised, high-level terms, and do not provide a detailed, accountable, and predictable definition of a single organisation's production process. By defining such a thorough development process model, that organisation's process becomes predictable, allowing the precise state of software projects to be easily ascertained. This, of course, increases the accuracy of project estimations in terms of future progress, time and human resource, hence positively affecting cost estimations.

As quality control becomes more of an issue in software, so it becomes more of an issue in its development. A quality development process is inevitably more likely to produce quality software consistently than an ad hoc development process. But how may we measure this quality? By characterising the maturity of an organisation's software process on a predefined quality scale, this quality becomes quantifiable. This work achieved by the Software Engineering Institute (SEI) at Carnegie Mellon University in America has provided a significant contribution to defining terms, characterising development quality, and offering a guideline in assessing an organisation's software process, as has the ISO 9000 series of standards and the SPICE framework. An important property of these process assessment methods is that they do not attempt to mutually exclude each other; the authors perhaps aware of the dangers of introducing too many non-compatible methods which may cause unnecessary confusion. The SPICE effort, for example, is designed to support the ISO 9000 series of standards, and to provide integration with the SEI's Capability Maturity Model.
In summary, Altmann and Boyce [Altmann and Boyce 1996] say it best:

Software and systems commonly fail to meet their purpose, not because of a lack of technical expertise on the part of the analysts and developers, but because of the failure of all stakeholders to comprehend the social, behavioural and political aspects of the user environment.

The next chapter will address how this understanding and awareness of an organisation's software process may be achieved by capturing and enacting that organisation's process.
This chapter will introduce and explain process formality, and how this process formality is currently being represented. There are two distinct approaches to representing the software process as a process model; graphical notation and process programming. They provide a defined (cf.: Capability Maturity Model, section 2.4) method of representing how a (family of) software system(s) should be developed. Two graphical notation approaches will be covered in section 3.2, and section 3.3 will introduce textual representation of process models. Section 3.4 will discuss the issues associated with deviating from a defined process model.

3.1 Classifying Process Modelling Approaches

McChesney [McChesney 1995], attempting to classify the various Software Process Modelling Approaches (SPMAs) comprehensively, presents a classification of different modelling worldviews; each approach can be additionally categorised by its inherent domain perspective. The choice of worldview greatly influences the choice of modelling notation, which can be graphical or textual: some approaches lend themselves only to one form of notation, and others to both. The Artificial Intelligence (AI) worldview, for example, would require textual representation since it concerns the application of AI-type programs.

- **Procedural Program Worldview**: Software process programs designed, implemented and maintained as a software system.

- **Artificial Intelligence (AI) Worldview**: Exploiting those domains for which AI-type programs can be applied, the process is considered to be a knowledge-intensive process performed primarily by intelligent agents who actually determine the optimum development sequence according to their experience, (implemented as an expert system), and local and global circumstances.

- **Database Worldview**: An entity-based view which defines a process with respect to project artifacts, such as documentation, requirements specification, etc., and operations which can be performed upon those artifacts.
• **Discrete Mathematical Worldview**: The process is described in terms of discrete mathematical structures. By using mathematical and logic-based models, verification of process characteristics such as correctness and reliability is possible before process execution.

• **Concurrent System Worldview**: This worldview advocates the software process as a highly concurrent and non-deterministic system. This process is characterised by synchronous and asynchronous events, constrained by concurrence, precedence and frequency. Model behaviour (liveness, deadlock, etc.) can be predicted with suitable formalism.

• **OR Worldview**: The values and relationships between process variables are described by mathematical equations. Characteristics of the process can emerge through simulation and analysis.

• **Socio-Linguistic (Socio-Technical) Worldview**: Based on the role activity model, roles achieve their individual goals through participation in language-based interactions according to established conversational patterns and rules.

However, the more esoteric perspectives aside, it is the *Procedural Program* and *Socio-Linguistic* worldviews that have had most influence on the research included in this thesis.

### 3.2 Graphical Representation of Process Models

The Role Activity Diagram (RAD) and the Unified Modelling Language (UML) sequence diagram notations will be detailed here, both of which are socio-technical.

Each section dealing with these approaches will also elaborate on how they may be applied via an example process. These examples give us insight into how process modelling approaches in general may be applied to situations.

#### 3.2.1 The Role Activity Diagram

The Role Activity Diagram (RAD) is the principle notation for STRIM (Systematic Technique for Role & Interaction Modelling), which has arisen from work done at Praxis on the modelling and analysis of business processes [Ould 1995]. This modelling method can be adopted for either incremental process improvement, or radical process re-engineering.
Ould lists eight laws for process modelling, capturing the general needs of the modeller:

1. *If you must have abstractions, make them concrete abstractions.* Any process modelling notation must deal in concepts that people relate to. Otherwise, how can they tell if a model's right?

2. *The real world is messy.* The notation must be able to model mess when necessary. Muddle modelling is the norm, not the exception.

3. *A model must mean something and only one thing.* If your model is ambiguous, how can you tell what it is telling you?

4. *Process models are about people, and for people.* The notation must make sense to people. If you can't explain the model in ten minutes, it doesn't make sense.

5. *There's what people actually do, and what they effectively do.* These are different and we must be able to model both.

6. *People do processes, but they work in functions.* These two mess each other up. The model must capture both - and the conflict.

7. *It's what people do, not what they do it to, that counts.* A process is about doing, deciding, and cooperating. Not about data.

8. *There are some basic business patterns.* These include the processing of a unit of work, plans, delegation, periodic activity, and contracts. We want to be able to capture them in our models.

Although law 7 may well be disputed from the standpoint of model execution, where *data* concerning materials (such as project deliverables) may well be necessary to describe the current state of an enacting process, these laws convey a no-nonsense perspective to process modelling. As implied by these laws, the notation adopts a socio-technical worldview, modelling people performing processes, not processes involving people.

Law 5 proves interesting; by modelling both what is *actually* done and what is *effectively* done, a greater overall understanding of the process can result. Of course, this distinction must first be acknowledged [Parnas and Clements, 1986]. This type of understanding would appear to be of great benefit to an organisation, especially to higher management, where it could prove more likely that a natural disparity between perceived activity and performed activity exists. By understanding both perspectives, process
improvement could be more productive; altering the effective model could result in a corresponding alteration in the actual model, which can be more easily applied in the real world.

The RAD notation itself models a process in terms of roles, role activities, and role interactions. A role can represent a single individual, a group of people, even an automated system. A role performs activities, and interacts with other roles socially. An interaction may involve the passing of a material from one role to another; for example, a Project Manager may pass an Initial Project Plan to a Requirements Analyst. A RAD example taken from Ould is given in Figure 3.1.

![Figure 3.1: An example Role Activity Diagram (RAD)](image)

This example models the design phase of a project. We have three roles with the names Divisional Director, Project Manager, and Designer, and each role contains a set of activities (denoted by black boxes), and interactions (denoted by white boxes). The Divisional Director role has a tick beside its name. This indicates that this role exists already as an instance. That is, a Divisional Director exists when this process model begins. Note that the definitions of each role do not just begin and end, they simply
continue (denoted by the ‘squiggle’ at the end and beginning of roles). Each role may have activities
defined before or after this part of the model, but we are not interested in them at this stage. The process
model here is depicting only the part of the overall process that we wish to examine.

The Divisional Director responds to an external event (denoted by the black arrow) which is triggered by
the occurrence of a project approval. This event happens outside the scope of this model. A new Project
Manager role is created to handle this occurrence, and the terms of reference (TOR) for the project are
agreed during an interaction between these two roles. The Project Manager starts a new Designer, writes
the TOR, and passes the TOR to the Designer. The Designer, receiving the TOR, then concurrently splits
his workload into two simultaneous tasks (denoted by the two triangles). One task is created to ascertain
which method to use, and the other to prepare and pass an estimate to the Project Manager, and receive a
project plan. If and only if both of these concurrent tasks have completed can the Designer continue. The
Designer produces a design, and carries out a design quality check. A decision is made (denoted by the
upside-down rectangles) as to the quality of the design. If the design is unsatisfactory, the design is
re-created (denoted by the thread path for the ‘N’ option rejoining above the ‘Produce design’ activity). If
the design is satisfactory, the Designer passes the completed design to the Project Manager, who produces
a debrief report. The final descriptor of the Project Manager indicates that he/she has reached a certain
state in the model; in this case, that the project has been completed and debriefed.

In this example each role only has one starting thread starting at the top of their boxes, but may have many.
Figure 3.2 illustrates how multiple threads can be modelled.

![Figure 3.2: A role with two separate threads](image)

Here two event threads are defined, and can be triggered and operate independently. One occurs when the
annual budget cycle begins, and the other when the monthly reporting cycle begins.

As will be seen in chapter 4, the RAD notation played an important part in deciding the approach and
features of Romula.
3.2.2 The Unified Modelling Language

The Unified Modelling Language (UML) [Rumbaugh et al. 1999] was developed as a successor to the object-oriented (OO) analysis (OOA) and design (OOD) methods that appeared in the late '80s and early '90s. It represents the culmination of ideas from various object oriented analysis and design sources, primarily from previous works authored by Grady Booch [Booch 1991], Jim Rumbaugh [Rumbaugh 1996], and Ivar Jacobson [Jacobson et al. 1992], otherwise referred to as the three amigos.

These authors argue against using formal methods to specify the UML notation rigorously. Although the formal specification of a method is itself totally unambiguous, (to the formal methodologist, at least), the value of this provability is not universal; there is no way to prove the mathematical specification actually meets the real requirements of the system. Although most OO methods are informal, this has not harmed their applicability. Many people still find them useful, and usefulness, he argues, is what counts.

Instead of defining UML formally, it is defined with reference to a meta-model, which describes the language usually as a class description. UML adopts a higher abstraction level than that of RADs. Above the actual UML language definition resides this meta-model. This meta-model describes the UML notation diagrammatically, illustrating associations and generalisation.

3.2.2.1 Application of the UML

UML is promoted as a 'language', (although visually a graphical notation), and not as a method, and is independent of process. For this reason it does not formally include a process which describes how to incorporate the language. However, a process framework is included as an example, and is shown in Figure 3.3.

![Figure 3.3: The UML Outline Development Process example](image)

The stages in this general process outline are described as:

- **Inception**: Business rationale and project scope initially established. Commitment from project sponsor is obtained.
Elaboration: More detailed requirements are researched and collected, high-level analysis and design are conducted to establish a baseline architecture, and a project build plan is created. A valuable activity conducted during this phase is the gathering of use cases, more commonly known outside the UML approach as scenarios. A use case is usually a short paragraph or three that describes a typical interaction between a user and the system.

Construction: In this process, this phase consists of many iterations; each iteration builds production-quality software, tested and integrated that satisfies a subset of the requirements of the project. The delivery may be external, to early users, or purely internal. Each iteration contains all the usual lifecycle phases of analysis, design, implementation, and testing.

Transition: Even this type of iterative process requires some work to be done at the end. This can include beta testing, performance tuning, and user training.

Another concept that is introduced is that of ceremony; the level of formal paper deliverables, formal meetings, etc. Bigger projects require more ceremony to ensure the project is progressing, although too much bureaucracy should, of course, be avoided, lest bureaucracy overwhelm the process of constructing software [Humphrey 1999].

3.2.2.2 The UML Sequence Diagram

The UML sequence diagram, a specific type of UML Object Interaction Diagram (OID), assumes a more generalised approach than that of a Role Activity Diagram. Although process models can be described in a socio-linguistic manner like RADs, UML sequence diagrams assume a more higher-level context. Not only are they designed to represent social behaviour between roles, the UML sequence diagram may also represent interactions between objects of any description. In addition, they include information which illustrates the relative time taken to perform tasks.

The sequence diagram is comprised of four primary elements:

- Objects which form part of the interaction scenario are depicted horizontally across the top of the diagram.
- The involvement of the object in the interaction over time is depicted vertically by a dashed line, the 'lifeline'.
'Foci of control' rest upon the lifeline depicting actions being performed by an object, and are denoted by thin rectangles. The length of the rectangle depicts the relative time taken for an object to complete an action. However, we may simply use a rectangle reaching from the top to the bottom of the diagram if this information is not important.

Interactions between objects are expressed with lines linking pairs of objects.

A simple example of a sequence diagram is given in Figure 3.4, although often sequence diagrams can be far more complex than this. We have included a simple example here, as the previous section which introduced the Role Activity Diagram has already substantially covered the aspects of a socio-technical notational model.

Here we can see three objects involved in this interaction scenario: the call initiator, (caller), the telephone exchange (exchange), and the call receiver (receiver). We can observe from this diagram how one might model the process of initiating a telephone call, with the various interactions depicted between each entity.

Clearly, these sequence diagrams bear much resemblance to RADs, although with sequence diagrams we can introduce time constraints into the model. This obviously makes it ideal for modelling real-time
systems, but we can forego this additional information and simply use the sequence diagram for modelling socio-technical interactions, as with RADs, if required.

3.3 Linguistic Representation of Process Models

The concept of process programming Osterweil [Osterweil 1987] is concerned with developing 'formal' specifications of software development processes. Section 3.3.1 introduces this concept, and section 3.3.2 presents a classification for such languages.

Comparison between different modelling approaches can prove difficult. With some more mature approaches, the process modelling language can form only a small part of the overall process modelling approach. Whilst some approaches offer just linguistic-based modelling, others offer complete process design methodologies; which is far beyond the scope of Romula (introduced in chapter 4) at its current state of maturity. In such cases, comparison will be made to aspects of each advanced, abstract methodology which are relevant to Romula. Another process modelling language, RolEnact [Phalp et al. 1998] which was also developed at Southampton University, will not be featured here, since a brief analysis of the language and comparison with Romula is given later in chapter 5.

3.3.1 Formality in the Software Development Process

Introducing strict formalism into describing software processes may yield substantial improvements in software quality. Osterweil [Osterweil 1987], in agreement with Lehman [Lehman 1991], highlights the importance of formality, introducing process descriptions, describing a software process as a "process program" in a computing language. Christer Fernstrom et al of the Eureka Software Factory [Fernstrom 1992] also emphasises the usefulness of expressing an organisation's process model as a process program. When enacted, process information concerning the process can be theoretically stored automatically. e.g. When process tasks were begun, times taken to complete tasks, etc.

Figure 3.5 describes an example given by Osterweil of a process program which describes a very straightforward type of process for testing application software. The example demonstrates some important qualities of process programs:

- Key aspects of the testing process are highlighted, whilst hiding lower level details.
- As such, it is a reasonable example of the use of modularity.
It conveys software process information clearly.

The use of a computing language, he claims, lends itself perfectly to describing software processes. Language features such as procedural abstraction and formal clarification do appear to aid in the understanding of software development and its various stages, forcing the writer of process programs to examine and clarify their in-house procedures more fully. Osterweil states many advantages to the process description approach, emphasising the potential of externalising them, expressing them rigorously, storing them archivally, and exploiting computing power to aid in guiding and measuring their execution. Such process programs theoretically could be compiled and even executed - hence validating and even providing control for those procedures, prompting appropriate team members (e.g. systems analyst, programmer) to fulfil specific tasks.

```
Function All_Fn_Perf_OK(executable, tests);
  declare executable executable_code,
    tests testset,
    case, numcases integer,
    result derived_result;
  --Note that executable_code, testset and derived_result are
  --all types which must be defined, but are not defined here.
  All_Fn_Perf_OK := True;
  For case := 1 to numcases
  --This is the heart of the testing process, specifying the
  --iterative execution of the testcases in a testset array
  --and the comparison of the results with expected behaviour.
    derive (executable,
      tests[case].input_data,
      result)
    if not resultOK (result,
      testcase[case].req_output)
      then
        All_Fn_Perf_OK := False;
        exit;
      --Note that the process specified here mandates that testing
      --is aborted as soon as any test execution does not meet
      --expectations. This is an arbitrary choice which has been
      --made by the process programmer who designed this testing
      --procedure.
    end loop;
  end All_Fn_Perf_OK;
```

**Figure 3.5:** Example process program given by Osterweil in his paper *Software Processes are Software Too*

Benefits are obvious, notably that a software development environment could be centrally controlled by an executing process program which suits that individual development team's procedures. However, Lehman argues that, in general, informality is also required, claiming that an $E$-type development process (transformation from a non-formal model of a real world application to a formal representation of the programmatic part of a solution system) can never be wholly formal. Since the process itself cannot be formal in this case, it can never be wholly expressed by a process program, except perhaps only at more
abstract levels. A more fundamental concern that will be addressed in section 3.5 and later in section 4.1 is how compatible programming languages are with a modelling paradigm.
3.3.2 Classification of Process Modelling Languages

A classification of Process Modelling Languages (PMLs) is provided by Chen [Chen 1997, Chen and Tu 1994a, Chen and Tu 1994b] with the following categories of PMLs:

- **Procedural**: Such as Chen’s CSPL, are based on imperative styles similar to the programming languages of Ada or C++.
- **Functional**: Following functional programming styles, they provide activity hierarchies via clear top-down functional decomposition.
- **Rule-based**: Similar to procedural languages, yet with higher abstractive qualities, they provide declarative meaning to processes.
- **Goal-based or planning**: Involving mixed paradigms, a process’ rules can be constructed to form process constraints or process goals.
- **Triggered**: A triggered (or event-driven) PML responds to and acts on external events.

Whilst each language type provides distinct modelling advantages and perspectives, Chen objectively states potential disadvantages to the various approaches:

- **Procedural**: These tend to exhibit relatively low-level abstraction. It can be difficult to understand from a higher-level perspective what is going on.
- **Functional**: Functional are difficult to implement due to excessive backtracking. Here the method of implementation hinders definition of the model; developers need to understand a software process very well to prevent this phenomenon.
- **Rule-based**: This type of modelling approach requires good understanding of the overall process’ meaning before the rule sets can be constructed.
- **Goal-based**: These require good balance between language theory and practicality, lest the language be too complicated or abstractive to be applicable.
- **Triggered**: This type of process program appears difficult to construct due to the lack of a general activity model view. Complex process programs appear very difficult to construct using this paradigm.

McChesney [McChesney 1995] adds to this categorisation, notably with **AI/Knowledge-based, Object oriented, and Formal specification** types.
3.3.3 Interact

Interact (forming part of the Intermediate [Perry 1994] process support environment) is a goal-directed modelling language. It defines three different loci of control for an activity:

- **Implicit, Internal Locus of Control:** Namely, preconditions, postconditions, and obligations for an activity. These add an abstracted descriptive quality that other PMLs lack. It should be noted that this representation forms a partial ordering of tasks; activities are not defined in any definite order, but can only begin when their preconditions are met. In addition, activities can only be deemed complete when their postconditions have been met.

- **Arbitrary, External Locus of Control:** These are user-oriented enactment controls: activity elaboration, state restoration (handling of exceptional circumstances), and enactment scheduling (external constraint on either the beginning or completion of an enactment). This enables the user to specify the ordering of activities, but can only do so within the implicit, internal locus of control.

- **Explicit, Internal Locus of Control:** Features enactment commands, enactment control, and control generation. A most interesting aspect of enactment control specification is the capability to define enactment statements sequentially or arbitrarily. Arbitrary definition allows the enclosed statement actions to be performed in any order, concurrently or otherwise.

Interact takes a more declarative approach to process modelling than Romula (introduced in chapter 4), although it is clearly influenced by the UNIX shell language (for each statements on sets, etc.). Its informal method of defining concurrent, arbitrarily ordered enactment statements offers a flexible and realistic approach. By not imposing strict statement order, real-world processes can be more accurately modelled, since in some cases a list of activities may be specified where the order is unimportant.

In the example given in Figure 3.6, which details an activity concerned with integrating and evaluating a number of tools, we can observe the preconditions, postconditions and obligations in use. In order for the Integrate activity to begin, an initial precondition has to be met: that the Tool-Release-Board has approved a release of a set of tools. When this has occurred, the activity can begin.

Within the actual activity, familiar programming-based constructs exist, reflecting the explicit, internal locus of control. In essence, a shift in program style has occurred. At this level of detail the process is being modelled using a procedural program style. Each tool that is submitted is added to the product build,
and each tool that disrupts the build is rejected (\texttt{tool-rejected()}). If it does not, the tool is evaluated (using \texttt{Evaluate}), and either rejected or approved via \texttt{Await-Acceptance/Rejection()}. 

In the \textit{results} section of the definition, two sets of postconditions exist. One permits the approved set of tools to be released (using \texttt{tools-released()}), with no obligations to be performed, before the activity ends. The other constructs a list of the rejected tools, and an obligation is performed that specifies that each tool needs to be modified (using \texttt{modify-tool()}). Any specified obligations are performed by the enactors of the process before the activity can end.

```
activity Integrate ()
  preconditions { Release-Approved(Tool-Release-Board) }
  {
    for each tool t in { tool t | submitted(t) } until Current-Time == Deadline:
      <
        Determine-Dependencies(t, dependencies),
        let testset' = testset + t,
        Build( testset', result ),
        ( result == false, tool-rejected(t) ),
        ( result == true,
          <
            < for each person P in { person p | owner[t1] == p & t1 in dependencies }:
              bind Evaluate(t, t1) to P
            >,
            Await-Acceptance/Rejection(t)
          >
        )
      >
  }
results <
  ( postconditions {
    approvedset = { tool t | tool-approved(t) },
    exportset = exportset + approvedset,
    tools-released(exportset) },
  obligations { } ),
  ( postconditions { rejectset = { tool t | tool-rejected(t) } },
  obligations { for each tool t in reject-set: modify-tool(t) } )
>
```

\textbf{Figure 3.6:} Example of an Interact activity

Despite Interact's high-level declarative approach to expressing enactment statements, in general it adopts a more low-level approach than Role Activity Diagrams. To define the preconditions, postconditions and obligations would require a good, detailed knowledge of the organisation's process.
The motivation for creating a procedurally-based language is clear; most developers have been using one programming language or another for some time. Process modelling using a similar language seems a natural step. However, one could have reservations as to the appropriateness of using such languages and associated semantics for modelling purposes, since it seems they implicitly force a procedural program worldview on the modeller. McChesney states for such approaches, this worldview presents potential problems and pitfalls, but unfortunately does not elaborate on precisely what they are.

CSPL [Chen 1997, Chen and Tu 1994a, Chen and Tu 1994b] adopts a distinctly low-level approach, using Ada95-like syntax. This appears to be a good choice, since Ada incorporates concurrency as standard, allowing easy concurrent specification. The language is object-oriented, which facilitates powerful handling of materials within a process model. Full material state is therefore inherently catered for. In addition, relationships can be specified between such objects, so, for example, if a requirements document was modified, a dependency link with the appropriate design document would trigger the appropriate updating of the design document. A CSPL example is given in Figure 3.7.

```ada
package design is
  type design_type is new DocType with record
    descriptor : string;
  end record;
  procedure modify(design_doc: in out design_type; req_doc: in req_type);
end design;

package body design is
  procedure modify(design_doc: in out design_type; req_doc: in req_type) is
    1 designer edit  design_doc referring  to req_doc using  editor;
    inform  design_and_source to set  design_modified;
  end
end design;
```

**Figure 3.7:** A CSPL definition of a modify design activity

In this simplified example, firstly a specification of the design process is given, detailing the new data type used (`design_type` new type of `DocType`), and the procedure used to perform design. The procedure takes a `design_doc` of type `design_type` and `req_doc` of type `req_type` as input. `design_doc` is also to be output, which is denoted with the additional `out` attribute. The `package body` then defines the actual procedure `modify`. It specifies that one (1) designer is to edit the `design_doc` whilst referring to the `req_doc`, and must use an `editor`. After this has occurred, `design_and_source` is `informed` so that they set `design_modified`. `design_and_source` is a relation which ensures that when the design is modified, the source code is appropriately modified to reflect the changes in design. Because operations can be defined on project artifacts (`req_doc`, `design_doc`) the language also adopts a database worldview.
Although an attempt has obviously been made to ensure that CSPL is readable as well as executable, it is not clear from the large example CSPL process program what exactly is happening. Many differing types of definition (package, role, procedure, function, relation, etc.) do not help to clarify how the process program operates, and most of these definitions seem to have too much basis in the realm of programming to be easily understood from a modelling perspective. For example, CSPL attempts to incorporate a multi-user facility, by assigning processes to roles (as demonstrated in Figure 3.7). However, an overall role activity view is not at all clear from a process modelled using this language.

3.3.5 PML

By using the ISPW-6 (6th International Software Process Workshop) example process model, a description is given on the ProcessWise Integrator [Robertson 1994] methodology. It addresses five model perspectives: activity, User, Organisation, Product, and Tool. The activity (or application) model utilises a language called PML for its process description purposes. It incorporates an interesting GUI sub-interface language which allows the modeller to define completely the user interface for each process model. The language is object-oriented and role-based, and is essentially dynamic; it allows changes to the process model ‘on the fly’, which provides extreme modelling power. Roles are defined by specifying classes, associations, resources (including variables and GUI definition), and role actions. PML forms only part of the overall model architecture which is defined as follows:

- **User Model:** This defines sociological behaviour within the model using various approaches, such as RADs. Hence this defines a socio-technical perspective.

- **Environment Model:** To make process enactment meaningful in the context of an organisation, certain roles may need to be defined which encapsulate certain services. In the ISPW-6 example process implementation, a *Library* role is defined which models the ability of other roles to access data contained in its system.

- **Organisation Model:** This simply illustrates the hierarchical organisation (in terms of responsibility) of the roles in the overall model.

- **Product Model:** This defines the product in terms of units each of which is composed of materials such as requirements, design, object code, source code, test plans, etc.

- **Activity or Application Model:** The model which is ultimately enacted, and is defined in PML. Application models reflect a process perspective; roles are defined to model task activities not role activities.
An example is shown in Figure 3.8 which illustrates some PML which forms the user model of an implementation of the ISPW-6 Process Example [Kellner et al. 1991] developed at the 6th International Software Process Workshop in 1991.

This class definition describes the essential behaviour for all 'task roles' within the implementation for the ISPW-6 process example. The term 'task role' is used to describe a role which is part of the application model, and actually defines process behaviour not role behaviour. The behaviour modelled is to display when required certain task information (identified as a_scheduleDates); to be able to display an appropriate message (identified in displayMessage) if the task role was altered by a higher authority task role (e.g. Project Manager, or PM); and to dispatch a message to the PM on conclusion of the task (sendIt).

WPRole isa Role with
classes
  assocs
    doneGP : giveport DonEnt
  resources
    wpEnt : WPEnt
    thisRole : String
    whoiam : String := ''
    doneEnt : DonEnt
    kill : Bool := false
    schedFormat : collof UIField:=collof UIField(
      UIField(name='startDate', label='Schedule Start'),
      UIField(name='duration', label='Duration'),
      UIText(name='requirements', label='Requirements'),
      UIField(name='notes', label='Note'))
  seemessage : Bool := false
  onHold : Bool := false
  messageString : String := 'New Scheduling Information'
actions
  a_scheduleDates:{
    UserAction(agendaLabel='Task Information');
    ModifyNow
      {agendaLabel='Task Information',
        object = wpEnt, readOnly=true,
        format=schedFormat}
  }
  displayMessage:{
    ViewResourceNow
      {agendaLabel=' Message', object=messageString, label=whoiam};
    seemessage:=false
  }
  when seeMessage
    sendIt:{
      Assign(from=whoiam, to=doneEnt.task);
      Give(interaction=doneGP, gram = doneEnt);
      kill:=true
    }
  when nonnil doneEnt
  termconds kill
end with

Figure 3.8: PML segment which forms part of an implementation of the ISPW-6 process example
This class definition is then used to instantiate sub-classes of this definition. For example, an instantiation of the *Modify Code* task role, (*MSCRole* - which is not defined in the paper), is a subclass of *WPRole* which defines additional activities necessary to model the *Modify Code* activity. Thus, a network of such task roles essentially defines a task network; interactions represent project materials passing between these task roles. *Source Code*, for example, passes from the *Modify Code* task role to the *Modify Unit Test Package* task role.

Also supported is the ability to model system 'housekeeping' activities (forming part of the environment model). Therefore, PML may be used to model automated *system* behaviour as well as role and task behaviour.

The paper suggests the ISPW-6 process example to be a good vehicle for appraising alternative process modelling technologies.

### 3.3.6 APPL/A

The Debus-Booch prototype, introduced by Song and Osterweil in two of their papers [Song and Osterweil 1998], is a more precise and complete implementation of the Booch Object-Oriented Design [Booch 1991] (BOOD) Software Design Methodology (SDM). Although SDM's are intended to be used to help design software more systematically, they claim that:

'[BOOD] is far too vague to provide specific guidance to designers, and is too imprecise and incomplete to be considered as a fully systematic process for specific projects.'

For these reasons, they applied the process programming concept to the design process. The process programming language utilised for this purpose is APPL/A. Unfortunately, since both papers focus primarily on the discussion of the enhanced BOOD definition, which is beyond the scope of this thesis, adequate explanation of the APPL/A language is not given. However, from the scant overview of APPL/A, and an example which is given in Figure 3.9, it can be ascertained that APPL/A exhibits the following features:

- **Object-Oriented**: In line with the BOOD methodology, the language supports classes, inheritance and methods. The example definition inherits *candidate_rel* of class *class_template*.
- **Event Driven**: Since the language supports triggered operations, (defined using the *trigger* command), it is event-driven.
Procedural Style: Although APPL/A appears predominantly event-driven, the language exhibits various procedural characteristics at the lower levels of definition, utilising imperative-style programming constructs, such as the case statement.

```
with candidate_rel, class_template;
trigger maintain_candidate;
   --| maintain the product of step 1
trigger body maintain_candidate is
begin
loop
 trigger select
 upon candidate_rel.update
 (name : string;
  needed : boolean;
  kind : candidate_type;
  update_name : boolean;
  new_name : string;
  update_needed : boolean;
  new_needed : boolean;
  update_kind : boolean;
  new_kind : candidate_type)
 completion do
 if needed = TRUE or update_needed = TRUE then
  --| change management is necessary only when
  --| candidate is selected or being updated.
  case kind is
   when class =>
    ...
   when abstract_class =>
    ...
 end case;
end upon;
...
```

Figure 3.9: Part of an example APPL/A definition

Covered in more detail in this paper is a section which stresses the advantages of detail in process definition. It gives a summary of design issues involved in translating process designs into executable code: Step selection, Refinement selection, Control condition selection, Control flow selection, and Concurrency specification.

3.4 Process Modelling: A One-Lane Highway?

This section discusses the issue of when deviating from a set process model may be necessary within an organisation, the consequences for doing so, and how this may be achieved safely.

3.4.1 Reasoning about Process Deviation
Once a process is defined as a process model, that process becomes predictable. As long as the process definition is adhered to, the organisation enacting that process is itself predictable. However, is this adherence always possible? To answer this question, one simple fact concerning human behaviour must be understood: humans are fallible. It is entirely possible that one (or both) of the following possibilities can occur:

- **The Model of the Process is Flawed**: Due to human error the process model is simply not representative in one or many areas of the actual process being enacted. Perhaps the model has been defined for some time and is being strictly followed, but has not been correctly updated to properly represent a continually changing (hopefully improving) development process.

- **Enaction of the Process Model is Incorrect**: Again due to human error the process model, whilst being accurate and representative in this instance, is not being correctly followed. An important step may be omitted, for example, that is critical to the success of the software project.

Even more disturbing, deviation from the process model could actually be necessary. The following is a real possibility:

- **Time Constraints Limit Adherence to the Model**: Due to a project being significantly behind schedule, it may be necessary to deviate deliberately from the model by omitting a step considered to be relatively unimportant. The CMM classes such activity omission as a property of an 'immature' software organisation (see section 2.4).

If any these possibilities become a reality, ad hoc process improvisation may be necessary to ‘fix’ the situation. Of course, by doing this, the process at this stage is no longer predictable.

### 3.4.2 Possible Consequences of Process Deviation

As previously discussed, the primary concern of deviating from a process model is that the process becomes no longer predictable. The following lists other possible consequences of process deviation:

- **Invalid Project Estimates**: Estimates that have been based on adherence to the process model may become, to a varying degree depending on the nature of the deviation, inaccurate.
Disruption of Inter-Personnel Dependency: Organisational entities (such as departments) that depend on other entities within the organisation can suddenly become unsure of the accuracy of the process model depicting those entities. If one such entity deviates from the model, unexpected events may result which disrupts their relationship with other entities, which subsequently disrupts the efficiency of the development process.

The consequences of process deviation should not be underestimated. Consider an organisation that has a defined process model that is usually strictly adhered to, and consider what happens if one organisational entity decides to deviate from that model (accidentally or otherwise) by omitting a step which normally produces an important artifact required by another entity. The repercussions of this occurrence may not be realised by the entity which produces that artifact, but that artifact may actually be crucial to the receiving entity, whose process greatly depends on that artifact. In this instance, because the receiving entity may not be able to complete its process successfully, other entities may be adversely affected in a similar manner, creating a possible cascade reaction throughout the organisation's process. It is conceivable that such an eventuality could adversely affect the quality of the process' final product, and hence significantly affect the success of the entire project. In effect, the advantages gained by achieving definition of a process may subsequently be lost.

3.4.3 A Methodology is Required

This discussion does not aim to promote deviation. But, as previously mentioned, the possibility exists that it can occur accidentally or as a necessity. This author believes that strict guidelines should be followed to minimise project risk should deviation occur. The following is a simple set of possible guidelines which detail possible types of deviance and the procedures that could be followed if this occurs:

- **Skipping Tasks**

  Recorded justification:
  
  - Reason for skipping the task.
  - Perceived consequences of omitting the task.
  - Estimated process model deviation-effect factor.

- **Inclusion of Extra Task**

  Recorded justification:
By estimating the effect the deviation will have on the process model by attempting to quantify it, the 'purity' of a phase enacted in the process model can be calculated by somehow adding these deviances together. The resultant factor gives a relative idea of how well conducted the process model was. However, it should be noted that it is crucial to the effectiveness of this approach that these deviations become as widely known as possible within the organisation. Only by being aware of these deviations can the organisation attempt to improvise a solution appropriately. An actual method for calculating such a factor is not given, and therefore is a topic for future investigation.

Further methods of deviation factoring could include methods of specifying maximum factor deviations for specific areas (or even individual tasks) within the process model, identifying the criticality of certain tasks. Further rules could be created to devise a more effective process model if the deviation factor for a particular phase is unacceptable; in this way we can specify (only as a whole) how deviant a particular phase is allowed to be. For example, time-critical application process models might require an extremely low maximum deviation factor (if any) to ensure that, as realistically as possible, the application produced has been created using a very tight and controlled process model (hopefully implying a more reliable quality product).

3.4.4 Discussion

Deviation can prove hard to factor into process models. Attempting to contain all possible real-world process eventualities within a model is clearly impossible, as we can only capture what we have become aware of from past experience, and can imagine might happen. In such situations where the process cannot
cope with the demands made from current events, or simply breaks down, improvisation then becomes necessary. It is unrealistic to expect any process model to be capable of handling any eventuality, and it is how well an organisation can execute decisions on an improvisational basis that defines how well it is able to cope with the truly unexpected. However, to some degree we can provide pre-defined provision for such eventualities by establishing a *meta*-process. Chapter 8 introduces a framework for integrating a software organisation's process with such a meta-process which attempts to provide arbitrary control over relations between sub-contracting organisations, and organisations to which that organisation is contracted to.

### 3.5 Summary

We have introduced some example approaches of how detailed models of a software process may be captured using either graphical or textual representations, and also introduced a classification scheme which categorises these approaches by their modelling worldview. These modelling approaches build on the generalised paradigms of software development (discussed in section 2.3) by allowing much lower levels of process detail to be specified.

Ould [Ould 1995], the author of the The Role Activity Diagram (RAD) approach, puts forward some laws for process modelling. In reference to section 3.2.1, law 8 proves most interesting:

> There are some basic business patterns. These include the processing of a unit of work, plans, delegation, periodic activity, and contracts. We want to be able to capture them in our models.

If patterns exist within business processes, surely it would follow that software processes, being a subset of business processes, would also contain these patterns, and perhaps other more specific patterns as well. This, of course, is certainly the case. In some way or another, every software process follows the requirements, design, implementation and testing ideology illustrated in the generalised process models introduces in section 2.3. Perhaps more interesting is that software process models share common deliverable elements, such as the requirements document, software design, and the implemented system. Could such common behaviour be exploited to our benefit? In section 4.7 we will introduce the concept of a *materials* abstraction: a possible extension to the new process modelling language introduced in the next chapter is introduced which attempts to exploit these similarities.

Many process modelling languages adopt syntax and associated semantics which have much in common with imperative programming languages. CSPL, for example, is a language based on the Ada95 Ada standard and the UNIX shell, which enables relevant features of each (Ada's built-in concurrency structure) to be used in specifying process models. By mapping modelling-based concepts found in process
modelling to programming-based concepts found in programming languages, process models become easily executable. In many ways, programming language concepts are quite compatible with process modelling: iteration, condition testing and concurrency, (e.g. in Ada), for example, are semantic features found in each. It seems logical, therefore, to use an existing programming language grammar as a basis for a process modelling language. However, what such approaches effectively achieve is the creation of a lower level of representation between the actual model and the real world, as illustrated in Figure 3.10.

![Levels of representation between the real world and the linguistic model](image)

As discussed previously, the conversion between the concepts of process modelling and programming is relatively straightforward. But is it necessary, or even desired? Imperative programming languages are designed to enable a programmer to describe how a computational task is to be achieved. However, process modelling is far more abstract. The level of detail and level of completeness of a model are both controllable properties: the model may be concrete or abstract, and it may define an entire organisational structure, or only a portion of interest. This is certainly not true of programming languages. To fulfil a task, the program must be at a set level of abstraction and fully complete. These provisos simply do not apply to process modelling. In addition, the low-level nature required by these languages (notably CSPL) to specify a program could prove a hindrance when specifying process models.

Would it not be better to simply define the syntax and semantics of a textual representation in terms of actual process modelling concepts and attempt to completely side-step the need to define a model using programming language concepts? Of course, the nature of the language would be dictated by the chosen worldview (section 3.1) and the chosen language style (section 3.3.2). The next chapter introduces a process modelling language which, as one of its objectives, attempts to address this issue.

48
This chapter will introduce a new process modelling language which is aimed at overcoming the shortfalls of existing process modelling languages identified in chapter 3. Section 4.1 will give an overview and brief discussion of those shortfalls, and suggest a possibly better approach to defining a new process modelling language. Section 4.2 details the requirements of this new language, and section 4.3 will present an outline of an approach to meet these requirements. Section 4.4 will detail the initial language definition, discussing the issues of implementing such a language, and section 4.5 will introduce the prototype implementation. The methods and results of verifying this language are presented and discussed in section 4.6. Section 4.7 discusses how the language may be extended to encompass data, or material, modelling. Other applications for which this language was found to be useful is given in section 4.8, and a discussion of whether the language meets its requirements is given in section 4.9.

4.1 Towards an Alternative Solution

As discussed previously in section 3.5, and demonstrated in sections 3.3.3 to 3.3.6, many process modelling languages are predominantly programming-based. Whilst providing a foundation upon which modelling concepts can reside, this approach inevitably means that such process modelling languages inherit the low-level, syntactically complex properties of computer programs, neither of which are desired in process models. Process models are about people and for people, (see section 3.2.1 for Ould's laws of process modelling), they are not primarily for computers. Although responsible for introducing the concept of the process modelling language, it is to some degree unfortunate that process modellers fell to programming languages as a way of executing their process models. Immediately, there is a conflict of objectives. Programs do tasks, but process models describe tasks.

A better approach would be to create a process modelling language that utilises modelling-based concepts in preference to programming-based concepts as much as possible, and adopts an appropriate modelling-based syntax. A language which achieves both of these goals could facilitate easier use by modelling professionals who may not have a technical background. Thus, we would be effectively reducing the level of programming expertise required to use such a language. Moreover, by ensuring the
syntax of the language is as close to natural language as possible, a model definition would be more readily readable and understandable to other non-technical staff.

4.2 The Requirements of Romula

Much effort has been applied to designing a process modelling language which can linguistically describe a RAD. Since we are by implication creating a socio-linguistic language, with which to model and understand development processes, the language was named Romula (Role-Oriented Modelling and Understanding LAnguage). The requirements of this language were as follows:

1. Allows a capability-to-capability correspondence between the RAD notation and Romula. This would promote easier translation from RADs to Romula.

2. A RAD described using Romula must be revertible to its former RAD state. If this can be achieved, then traceability exists between the two modelling approaches.

3. Romula is a viable process modelling method in its own right, in that models can be constructed using Romula alone. This is elaborated as follows:
   
   3.1 Romula must be syntactically concise; models created using Romula should contain a minimum of syntactic 'glue'.
   
   3.2 Romula must be semantically clear and syntactically readable, promoting easier creation and understanding of process models.

In order for the first two of these requirements to be fulfilled, the process modelling capabilities of the RAD notation as outlined in Section 4.3 became linguistic issues which needed to be resolved. The third requirement is also a key one: the aim is to produce a modelling language that is predominantly modelling-based and not programming-based, in that the language embodies only relevant programming principles where such features are required.

In addition, it should be noted that Romula was never envisaged to be anything more than a process modelling language and animator; rather, it was designed to be integrated into a larger suite of modelling tools, which together would form an Integrated Process Support Environment (IPSE). This possibility is discussed in the conclusions in chapter 9.
4.3 Outlining an Approach

As described in section 3.1, there are many possible worldview approaches to modelling a process. Conceptually, two of these proved to be the most fundamental, in that they each offer an opposing method of process representation:

- **Procedural Worldview (Process-Oriented):** The model describes the various activities involved in a process. This method may or may not model the activities of the people, or groups of people, involved in the process. This method offers what could be termed *process clarity*, since we are modelling *process activity* at the top level.

- **Socio-Linguistic Worldview (Role-Oriented):** The model depicts the activities of those people, or groups of people, involved in a process. This method offers what could be termed *role clarity*, since we are modelling *role activity* at the top level.

Of course, since it was required that the language Romula allowed straightforward translation between itself and RADs, which are socio-linguistic, a socio-linguistic approach essentially has to be adopted. The considerable advantage of this was that socio-linguistic process modelling languages are relatively new. However, the procedural worldview also proves interesting, and section 4.8.3 introduces a possible opposing approach to the one adopted by Romula.

In addition, it was envisaged that adopting this approach would further the current state of research for this type of process modelling language within the Declarative Systems and Software Engineering research group of Southampton University. By creating a new process modelling language separately to that of RolEnact [Phalp et al. 1998] (a previous socio-linguistic modelling language developed within the DSSE group), improvements, contrasts and necessary similarities between the languages may further the understanding of this approach to process modelling.

Ould's RAD notation [Ould 1995] has proved to be most insightful in determining the necessary capabilities for Romula. Ould's notation allows a process modeller to deal with:

- Alternate (condition-based) paths of process execution.
- Concurrent paths of execution.
- Collaboration between multiple roles.
- Instantiation of roles as and when they become necessary.
• External events.

• Multiple threads within a role which are activated by external events.

Thus, Romula would clearly have to be able to represent these modelling constructs in order to represent RADs. Each of these features became linguistic issues that had to be resolved within Romula, and are discussed in section 4.4.

The choice of linguistic style was crucial. The style had to be easily understandable, and offer the ability to model RADs effectively. The RAD notation offers a clear, easy to understand way of representing process models; RADs clearly represent the actions of each of the roles throughout a process, and in order to provide straightforward RAD-to-Romula translation, and vice versa, this clarity had to become an inherent part of the language. Hence, by ensuring that Romula assumed the same level of technicality as RADs, those that could understand a RAD model should be able to understand a Romula model.

4.4 The Initial Romula Language Definition

This section details the initial Romula language definition. As Romula matured, extra features were added and these will be introduced at appropriate points later in this chapter and elsewhere in this thesis.

4.4.1 Concurrency

This problem proved to be the largest and most time-consuming. The problem was representing multiple threads of execution in a language, since the difficulty arises when the threads become recombined. Figure 4.1 shows an example of how concurrency is demonstrated in a RAD, and the notation extension created to solve the problem. Note that we have deliberately used nested concurrency to provide added complexity to the example.

![Diagram of concurrency in RADs](image.png)

**Figure 4.1:** An example of how concurrency is
demonstrated using the RAD notation

As demonstrated, in a RAD one thread of many concurrent execution threads may join at any stage in any other thread. Therefore, each thread needs an identity so it can be referenced. Figure 4.2 shows how this example is expressed in Romula, assuming it is embedded within role 'Test'.

```plaintext
ROLE 'Test'
{
    CONCURRENT
    TASK 'doThis' :
        DO 'A'
        EXPECTJOIN FROM 'Test.doThat.a'
        DO 'E'
        EXPECTJOIN FROM 'Test.doThat.b'
        ENDTASK
    TASK 'doThat' :
        CONCURRENT
        TASK 'a' :
            DO 'B'
            JOIN TO 'Test.doThis'
            ENDTASK
        TASK 'b' :
            DO 'C'
            EXPECTJOIN FROM 'Test.doSomeThing'
            DO 'F'
            JOIN TO 'Test.doThis'
            ENDTASK
        ENDCONC
        ENDTASK
    TASK 'doSomeThing' :
        DO 'D'
        JOIN TO 'Test.doThat.b'
        ENDTASK
    ENDCONC
    DO 'G'
}
```

Figure 4.2: How Figure 4.1 is expressed in Romula

By using a Process.SubProcess (or even a Process.SubProcess.SubSubProcess) notation, concurrency can be nested within concurrency. One problem with representing concurrency in a language is that multiple threads of execution can quickly become complex. Therefore, instead of a straightforward Label command (as in Pascal or assembly language), the use of ExpectJoin and its associated Join command is more intuitive and process modelling specific. In addition, since the ExpectJoin command explicitly specifies from which thread execution will be Joined, backtracing of threads is easily possible. Hence this method of representing concurrency allows multiple threads of execution to be better followed and understood, and ensures that ambiguity when executing the model on an interpreter is impossible. It must be noted that an ExpectJoin causes that thread to enter a wait state until the corresponding Join from another thread is reached to synchronise the join.
Note that commands following a *Concurrent* structure are not executed until all threads within that structure have been completed. Completion of a concurrent task can occur by either reaching an *EndTask* concurrent task terminator, or *Join*-ing another concurrent thread, in which case the *Join*-ing thread is terminated. Figures 4.3 and 4.4 clarify the nature of the *Concurrent* structure.

![Figure 4.3: Simple Concurrent example](image)

![Figure 4.4: Representation of Figure 4.3 to demonstrate continuance](image)

The thread of execution of Romula in Figure 4.3 simply rejoins the main thread after the *Concurrent* ... *EndConc* structure, thereby acting exactly as its RAD counterpart in Figure 4.4.

The *Do* statement in Romula is synonymous with the 'black box' in RADs. They both represent a task abstraction, the details of which we are not concerned with. However, it may become necessary later when refining a process model to elaborate on these atomic activities, by 'opening' the black box and replacing it with a detailed definition of how that task is conducted.

### 4.4.2 Interaction

This issue was relatively straightforward to resolve. See Figure 4.5 for an example.

![Figure 4.5: Example of interaction in a RAD](image)
As with the RAD notation, these types of interaction imply 'synching' of roles; i.e. each role halts execution until all other roles involved are prepared for the interaction to occur. As with the solution to concurrency, the language syntax for interactions is kept intuitive, clear, and self-explanatory.

This interaction is handled as demonstrated in the simplified example in Figure 4.6.

 ROLE 'Manager'
 { 
  INTERACT WITH 'ProjLeader' FOR 'Reason1'
  INTERACT WITH 'ProjLeader','Programmer' FOR 'Reason2'
  INTERACT WITH 'Programmer' FOR 'Reason3'
 }

 ROLE 'ProjLeader'
 { 
  INTERACT WITH 'Manager' FOR 'Reason1'
  INTERACT WITH 'Manager','Programmer' FOR 'Reason2'
 }

 ROLE 'Programmer'
 { 
  INTERACT WITH 'Manager','ProjLeader' FOR 'Reason2'
  INTERACT WITH 'Manager' FOR 'Reason3'
 }

Figure 4.6: Representation of RAD interaction in Romula

Since Romula is based on the RAD notation, Romula must adopt a role-based approach to process modelling. Figure 4.6 also demonstrates how roles are to be represented, although the solution to this is simply a logical extension to the language.

4.4.3 Conditional Execution

Initially, this was considered to be a simple problem. However, consider Figure 4.7.

Figure 4.7: Example of RAD conditional statement
Clearly, the result of the condition does not generate two separate threads, but simply selects a thread with which to continue execution. This raised the issue of how a single thread rejoins itself in such a situation. \textit{Join} and \textit{ExpectJoin} could not be used, since they involve a wait state. By examining Figure 4.8 we can see that execution would indefinitely pause at the first line; the corresponding \textit{Join} would never be reached. (Note that the \textit{ExpectJoin} thread reference is effectively referencing the role thread itself and not perhaps 'Test.YES' and 'Test.NO'; the \textit{Condition} statement does not spawn threads).

What was required was a variant of the \textit{ExpectJoin} and \textit{Join} commands, which allowed a thread to alter or to 'link' execution to another point in itself. Although semantically identical in to the infamous \textit{Goto} command in programming as noted by Dijkstra [Dijkstra 1968], Ould states that gotos are required to model a real-world process. In addition, we have adopted a careful syntactic strategy to help ensure that the modeller encapsulates meaning when using these statements. Usage of the syntax \textit{ExpectLink For} and \textit{Link To} is to force the modeller to use a reason (i.e. for the alteration in control flow) as the label, and not some arbitrary string of characters. See Figure 4.9 for a solution to the example in Figure 4.8.

\begin{verbatim}
ROLE 'Test'
{}  
  EXPECTJOIN 'Test'
  DO 'X'
  CONDITION 'VALUE > £10000'
  IF 'YES':
    DO 'Y'
    JOIN 'Test'
  ENDIF
  IF 'NO':
    JOIN 'Test'
  ENDIF
ENDCOND

ROLE 'Test'
{}  
  EXPECTLINK FOR 'Restart'
  DO 'X'
  CONDITION 'VALUE > £10000'
  IF 'YES':
    DO 'Y'
    LINK TO 'Restart'
  ENDIF
  IF 'NO':
    LINK TO 'Restart'
  ENDIF
ENDCOND
\end{verbatim}

\textbf{Figure 4.8: Loop example using} \textit{ExpectJoin} and \textit{Join}

\textbf{Figure 4.9: Loop example using \textit{ExpectLink} and \textit{Link}}

For \textit{Link} and \textit{ExpectLink} referential purposes, the single, main thread of execution is referred to as the 'role' thread. Another approach would have been to use the \textit{If ... Then ... Else} statement for simple choices, however, for syntactical simplicity and clarity the \textit{Condition} statement is used in such situations instead. Also note that \textit{Condition} statements can of course be nested (as with \textit{Concurrent} statements) if required.

\subsection*{4.4.4 External Events}

Incorporating the event aspects of the RAD notation was straightforward, and this allows a differing event-driven paradigm to be used to any degree if desired. See Figures 4.10 and 4.11.
Simply, if an event is triggered, the corresponding language segment is executed. Each event is treated as a separate thread. e.g. triggering event DoX starts up a thread Test.DoX which executes Do 'X' and exits. Interaction involving events is identical in operation to that of ordinary role code segments, and event threads can utilise Join and ExpectJoin in the same way as Concurrent Task threads do (see Section 4.4.1). Note that the main language segments (inside { and }) do not necessarily need to contain anything. We can therefore adopt a completely event-driven approach if desired.

4.4.5 Instantiation of Roles

A simple RAD role instantiation in Romula is shown in Figure 4.12. Since some roles are implied to exist before other roles which may become instantiated later, roles are created explicitly at the start of Romula. Each role creation implies a new thread is created for that role.
4.4.6 Handling Instances

One problem with RADs which is not really addressed by Ould is the problem of multiple instances of roles. Figure 4.13 shows two simple RAD roles involved mainly in communication with each other. (e.g. One Shopkeeper and eight Customers).

This is fine and unambiguous, until you consider the possibility of a number of Customer instances wanting to communicate with just one Shopkeeper. This model representation does not differentiate between them, and thus nor does the Romula representation given in Figure 4.14. Ould only briefly explains how instances should be counted, but does not explain how to differentiate between instances.

We have now also introduced the notion of the dominant role in interactions, which, in Romula, is expressed by stating which role is the submissive (SubInteract) one. This may seem contradictory, but it is
an implementation constraint which will be explained later in section 4.8.2. Essentially, during model execution, a submissive role cannot initiate an interaction. Only the role utilising the Interact (or dominant) type of interaction can achieve this. It should be noted that, by implication, all uses of Interact specify a dominant interaction by default. Where two (or more) role instances specify an Interact interaction, any one of those role instances one can initiate it. If two role instances were to specify a SubInteract interaction, both would not be able to initiate it, so the interaction would never occur.

Consider this example executed with, for example, two shopkeepers and two customers. It is conceivable that, without the proper controls, if the two customers interacted in complete synchronicity when dealing with their respective shopkeeper, a Romula execution tool could become confused with exactly which customer is communicating with which shopkeeper. This is a possibility, since this Romula definition does not state which instance of a given role is the target of each interaction. Thus, although Shopkeeper1 may begin a transaction with Customer1, it may end the same transaction interacting with Customer2. The straightforward solution is given in Figure 4.15. Assuming this were to be executed by a runtime interpreter, (see section 4.5), we could now instantiate as many Shopkeepers and Customers as we wished, by including an arbitrary number of Start Shopkeeper and Start Customer statements at the start of the definition prior to execution.

```
ROLE 'Shopkeeper'
{
    SUBINTERACT WITH 'Customer' FOR 'PurchaseRequest' TO '$customer'
    DO 'Locate Product'
    INTERACT WITH '$customer' FOR 'GiveProductToCustomer'
    SUBINTERACT WITH '$customer' FOR 'GiveChangetoShopkeeper'
}

ROLE 'Customer'
{
    INTERACT WITH 'Shopkeeper' FOR 'PurchaseRequest' TO '$shopkeeper'
    DO 'Get change ready'
    SUBINTERACT WITH '$shopkeeper' FOR 'GiveProductToCustomer'
    INTERACT WITH '$shopkeeper' FOR 'GiveChangeToShopkeeper'
}
```

**Figure 4.15:** Romula representation of Figure 4.13 which deals with instances

If we consider the Shopkeeper role, an instance of a Customer at the appropriate state for the first interaction is selected, and the interaction commences as before. However, this customer instance is placed into the $customer variable (which only has scope within the Shopkeeper role) for later referencing. Instead of further selection of Customers taking place at the next two interactions, the same Customer instance is referenced by using the $customer variable. A similar procedure occurs within the Customer role. This ensures that once an instance of a role is selected, no other instance of that role can 'hijack' any later interactions. Continuity and realism is therefore preserved within such models. It should be made
clear at this point that these are not variables in the strict programming sense. They represent role 'memory'. In reality, roles are aware of the individual identities of many instances of a role, and can distinguish between them. This feature allows us to model this memory and so better model reality. This is a good case of determining a need and adapting an appropriate programming-based concept to fit into a modelling-based language.

Another more profound advantage and extension of this method can be understood by considering a role which creates an arbitrary number of other roles, and at a later date is required to communicate with all of them. Perhaps for gathering reports from a number of task forces for the attention of a task force manager. Figure 4.16 shows the Romula representation.

```plaintext
ROLE 'TaskForceManager'
{
    EXPECTLINK FOR 'TaskForceCreation'
    CONDITION 'Create a new Task Force?'
    IF 'Yes':
        START 'TaskForce' TO '>$taskforces'
        LINK TO 'TaskForceCreation'
    ENDIF
    IF 'No':
    ENDIF
ENDCOND

    DO 'Some managerial task'

    INTERACT WITH '>$taskforces' FOR 'GatherReports'
}

ROLE 'TaskForce'
{
    DO 'Some task force task'
    DO 'Write a report on the task'

    SUBINTERACT WITH '>$CREATOR' FOR 'GatherReports'
}
```

**Figure 4.16:** Romula representation of a TaskForceManager and TaskForce hierarchy

The TaskForceManager can thus create a number of TaskForces, and then communicate with them to gather their reports. As can be seen, the Start command can assign instances to instance lists if required. The $>Taskforces notation simply denotes appending a role instance to a variable, so that this list of instances is used for referential purposes later. The $CREATOR instance reference is assigned at instance creation time, and refers to its parent, in this case the TaskForceManager.

Instance variables can also be passed to other roles. This functionality extends the types of model which can be created with Romula, such as roles which allocate instances of roles to other roles, like the barber shop example given in Figure 4.17. The barber shop model provides a satisfactory testing foundation for
socio-technical process modelling languages, and has previously been used in this capacity with RolEnact [Phalp et al. 1998]. In this example both the Barber and Customer check in with Reception, which logs both of the Customer and Barber instances with which it communicates. Using PassInstance, it allocates a barber to a customer (from $barber), and vice versa. As with the shop example detailed earlier in this section, we could instantiate as many Barbers and Customers as required to test the model.

```plaintext
ROLE 'Receptionist' {
    INTERACT WITH 'Customer' FOR 'CustomerChecksIn' TO '$customer'
    INTERACT WITH 'Barber' FOR 'BarberChecksIn' TO '$barber'
    PASSINSTANCE TO '$customer' FOR 'AllocateBarber' FROM '$barber'
    PASSINSTANCE TO '$barber' FOR 'AllocateCustomer' FROM '$customer'
    INTERACT WITH '$customer' FOR 'Payment'
}
ROLE 'Customer' {
    INTERACT WITH 'Receptionist' FOR 'CustomerChecksIn'
    GETINSTANCE FROM 'Receptionist' FOR 'AllocateBarber' TO '$alloc_barber'
    INTERACT WITH '$alloc_barber' FOR 'CutHair'
    INTERACT WITH 'Receptionist' FOR 'Payment'
}
ROLE 'Barber' {
    EXPECTLINK FOR 'CheckInWithReceptionist'
    INTERACT WITH 'Receptionist' FOR 'BarberChecksIn'
    GETINSTANCE FROM 'Receptionist' FOR 'AllocateCustomer' TO '$alloc_customer'
    INTERACT WITH '$alloc_customer' FOR 'CutHair'
    LINK TO 'CheckInWithReceptionist'
}
```

Figure 4.17: Example of an allocatory role, assigning barbers to customers and vice versa

Note that we could assume that this definition would require some alteration before multiple Receptionists could function correctly; we do not in either the Barber or Customer definitions capture the identity of the Receptionist whom they initially interact with. However, it could be argued that in reality, it does not really matter which Receptionist performs each of the tasks of checking in, allocation, and payment. Here we have an example of potential process uncertainty, (see section 2.2), since an assumption could be made concerning the process' domain which may prove incorrect. Therefore, further investigation into the practices of the Receptionist would be required.

GetInstance is used in Customer and Barber to receive these allocations, which are used as references for these roles to communicate for the CutHair interaction. The Customer then simply pays Reception (assuming there is only Reception instance, since we are not logging which Reception is being interacted with).
4.5 The Prototype Romula Animator

Originally, the prototype was to incorporate a RAD editor, which would allow RADs to be graphically designed and then converted into Romula. However, it was decided that this sub-project was a too time-consuming and complex problem which fell outside the main scope of process modelling, so higher priority was given to the Romula animator. The animator itself is composed of two separate parts: the Romula interpreter, (incorporating two intrinsic parts, the thread control handler and the language parser), and the graphical user interface. The following sections will detail each of these parts.

4.5.1 Evaluation of Prototyping Platforms

For the initial plan of the prototype, the chosen implementation language needed to meet the following requirements:

- **High-level:** To facilitate faster prototype development.
- **GUI-support:** To provide functionality for displaying the state of executing models, and to provide means for the user to manipulate execution of the model.
- **Performance:** An adequate level of performance would be required for displaying the model and handling multiple roles and role instances.
- **Interpreted:** An optional requirement which in most cases increases error recognition and correction (due to better error analysis and description) and therefore increases productivity during prototyping.

C and C++ (with various widget toolkits), Tcl/Tk, and Perl/Tk were considered as prototyping platforms. However, it was decided that the C variants are not adequately high-level, and would contribute to an unnecessarily lengthy implementation phase. Tcl/Tk, whilst high-level and providing an integrated GUI interface, is, in this author's opinion, lacking in computational flexibility. Perl/Tk meets the above requirements, and also provides an integrated GUI interface, high-level parsing and file handling features, (which would be ideal for interpreting Romula), and is easier to convert to a more concrete implementation in C than Tcl/Tk if necessary. In addition, the Perl/Tk distribution (at version 5.003 at last count) reached production-grade maturity at version 5, making it perfectly suitable for prototyping. An added major advantage of Perl/Tk is that it can be run on a Windows 95 or UNIX-variant platform.
4.5.2 Outline of the Implementation

Since Perl/Tk was used as the development platform, which deals with files, parsing, user interfaces, etc. at a substantially high level, it was possible to include the entire implementation within one single Perl executable. An overview of how the Romula implementation integrates with the operating system's windowing interface is given in Figure 4.18. For explanations on the thread control and user interface handlers see the next two sections.

![Figure 4.18: Overview of how Romula's User Interface Handler, its Interpreter (Thread Control Handler and Language Parser), and the operating system's windowing interface fit together](image)

At the highest level, the graphical windowing system of the host platform provides access to the functionality of the interpreter. Since Perl/Tk is a truly portable language, the Perl/Tk programmer does not need to concern themselves with the idiosyncrasies of the implementation platform itself. For example, the UNIX operating system and the X-Windows graphical interface are entirely separate processing systems, whereas the DOS operating system and Microsoft Windows 95 graphical interface are far more integrated. However, this operating system/graphical interface relationship is not visible to the Perl/Tk
programmer. Essentially, the ideal is that an application that runs on one Perl/Tk platform will run on another.

Romula's user interface handler reacts to events triggered by the host platform's graphical windowing system, which is handled by the Tk part of the Perl/Tk package. If an event is a role instance 'step' event, (initiated by selecting the 'Next' button on the user interface, see next section), this causes the thread control handler to call the interpreter's language parser to parse that role instance's next command in the Romula file. In effect, therefore, instead of the implementation parsing the entire Romula file and translating it into an internal representation for processing, the implementation utilises a 'parse-on-demand' approach. This offered the significant advantage of a simpler interpreter, since support for a complex internal representation was not required, nor was a Romula-to-internal representation translator.

Once the language parser has parsed a role instance's command statement, it then alters that role instance's thread state by informing the thread control handler which makes the appropriate changes. For example, when a role instance initiates a Concurrent command with two concurrent tasks, the thread control handler starts two separate threads, (associated with the initiating role instance), to handle each concurrent task. Once the state of a role instance has been altered, the thread control handler informs the user interface handler to reflect the appropriate changes in the user interface display. The creation of two concurrent tasks by the thread control handler, for example, would require two separate concurrent windows to be created by the user interface.

4.5.3 Process Control

The three types of process 'threads' in Romula are Role, Concurrency Tasks and Events. They form individual processes (possibly within the animator domain, not necessarily the operating system domain). As they are triggered at runtime, they each require a thread for their execution to be created.

Two possible methods were considered. The first required using operating system-dependent process management to deal with threads, and the second was to create a separate thread control handler within the interpreter, which associates threads with code segments of the types mentioned. The latter was chosen, since this gave far more flexibility and control over threads, and maintained platform-independence, or more specifically, operating system independence. This allowed us to maintain easy portability between the Windows 95 and UNIX-variant platforms.

The choice of an internal thread control system meant that interactions between threads also had to be handled internally. For the most part, this proved to be a straightforward process. By assigning an execution 'state' to threads, threads can be effectively 'paused' when required, a method used for both
interaction and concurrency. Threads can be 'paused' waiting for an interaction until other threads involved in the interaction were also ready to participate. When this occurs, (and the interaction completed), all these 'paused' threads can then simply be unpaused and allowed to continue. With concurrency, a role's main thread which has spawned two concurrent threads, for example, is paused until those two concurrent threads have completed.

4.5.4 The Graphical User Interface (GUI)

The GUI was designed around the thread control system. Each type of process is given a graphical window to interact with the user, which allows the user to step through each thread's code segments separately. In addition, a message window for Romula debugging purposes shows process information including occurring interactions and thread creation.

Only the basic human-computer interface (HCI) requirements will be given here, since a detailed study of HCI design rationale is beyond the scope of this thesis. The GUI requirements were as follows:

- **Semantic-level interaction:** GUI designed around the semantics of the system [Dix et al. 1993]; dialogue with the user to be based around the internal running of Romula.
- **Intuitive interface:** Important, since the system is centred around a concurrent-thread paradigm; each thread in execution must be fully controllable.
- **Analysis-based interface:** Since Romula animator is basically a task analysis system, and not a data creation system, the GUI had to be tailored towards clear, understandable data presentation.

The next sections will cover the various thread-type window designs: the message window, and the role, concurrent, and event windows which form the user interface.

4.5.4.1 The Message Window

Layout of the message window is shown in Figure 4.19. This provided an initial starting point for testing and debugging the mechanics of the interpreter, and therefore was not intended as an end-user interface. However, it can prove useful when debugging complex models as it gives an overall view of model activity.
Figure 4.19: Layout of the animator’s message window

The layout is simple yet descriptive. The left column contains either the thread name (enclosed in quotes), or an animator’s functionality manager (e.g. Interaction Manager). The right column displays the corresponding action of its initiator. Whilst the Quit button exits the Romula animator, Next group-steps all threads at once, instead of allowing the user to step individually through each thread.

4.5.4.2 The Role, Concurrent and Event Windows

Figures 4.20, 4.21 and 4.22 demonstrate each of the various types of window layout.

In each type of window some or all of these following types of sub-window are displayed:
• **Information**: Displays in either plain English or Romula each Romula non-empty line as it's executed.

• **Events**: If a role contains events that may be triggered, they can be executed from here at any time during the role's execution. Note that external events can only be triggered from a role.

• **Conditions**: As conditions are encountered they are displayed here for selection.

The buttons perform the following tasks:

• **Exit**: Pre-emptively end the thread. This theoretically could happen in a real-world situation. For example, a task force manager may wish to end a task force pre-emptively. Additionally, this feature may be required if a role component has been incorrectly defined syntactically, which may, for example, cause a state of deadlock within that role.

• **Mode**: Toggles between displaying thread execution information in plain English or in Romula (for debugging purposes).

• **Next**: Executes one Romula instruction when pressed, displaying the results in the *Information* window, unless an interaction is encountered (see below).

• **Exec**: Automatically executes Romula instructions within the thread, displaying the results in the *Information* window, until an interaction is encountered (see below).

With the *Next* or *Exec* functions, if an interaction is encountered within a thread, the interaction cannot occur until the other roles involved are also ready to enter the interaction.

### 4.6 Verification of Romula

This involved demonstrating that the Romula animator can successfully and reliably execute a process model coded in Romula. Adequate accomplishment of this objective required adopting the 'reverse psychology' testing method; prove that the Romula Animator was unreliable when executing these models. When faults were found with the animator, they were listed and the testing process continued, until the animator had been adequately tested. Following this phase, faults were fixed and the process restarted, until the animator reached an adequate level of stability. Use of the classification 'adequate' was deemed satisfactory when applied to the stability of the animator, since Romula was envisaged only as a prototype to demonstrate proof of concept.
4.6.1 Verification Method

This testing phase involved devising process models which tested all the features of Romula, each with the simple aim of crashing the animator, or otherwise rendering it unusable. This involved testing two aspects of the animator; the user interface, and the Romula interpreter. Since the interpreter was only a prototype, and because of time constraints, proper exhaustive testing of every Romula feature was infeasible.

Many of the process models devised were, strictly speaking, unrealistic and overly complex. This was to ensure that as many paths of execution as possible within the Romula animator were tested. For testing the user interface, the general theme of the tests was multiple roles, multiple events, and multiple conditions. Extensive usage of these features ensured that high demands were placed on the user interface.

For the testing of the interpreter, the theme of many of the tests conducted was multiple roles, embedded concurrency, and multiple interactions between roles. This approach was adopted to test what was believed to be the least stable and most complex aspect of the interpreter - the inter-role interaction mechanism.

Although verification is a fundamental and required activity for software development, examples of the test cases themselves will not be given here, since they represent a purely engineering-based activity not relevant within the context of this thesis.

4.6.2 Results of Verification

A contrived process model was developed in parallel with Romula for the purposes of validation (see chapter 7), and then tested on the Romula animator, and so verification to a large degree had already been conducted automatically. The complete test plan and results will not be detailed here, since verification is solely a mechanistic and purely engineering-based activity, but many minor faults were detected during this phase and fixed. As suspected, most of these faults were within the complex interaction mechanism.

4.7 Extending Romula's Capabilities

4.7.1 RADs and The Software Process

Although the RAD notation has the ability to abstract and simplify, Ould states that it is better to design a RAD modelling a complex situation completely than to oversimplify the model and produce an unrepresentative RAD. Now consider a CASE tool which executes a complex process model with
numerous roles and threads modelled using a RAD. The CASE tool graphically displays the points in each role and thread in the RAD to represent the state of the model in execution. Such a complex RAD may have many ‘model state points’ and it may be difficult, even with a simplified graphical display, to show the process model in an overall perspective.

The problem is that RADs represent a process model as a role-based abstraction, representing processes within a role. This gives role clarity, but in a complex model it is difficult to follow the overall processes; a model may have many roles involved in a single process (e.g. requirements gathering). Another approach would be to formalise and create process clarity, where the abstraction mechanism is a process instead of a role, representing the activities of roles within processes (like CSPL). Since the two are mutually exclusive, it is impossible to have both abstractions within a single model, and each have their disadvantages. However, by somehow incorporating process clarity in our RAD-based process modelling language, the advantages of each approach could become apparent.

The RAD notation was initially developed to model business processes, however, software development processes in the large always follow similar patterns. i.e. Requirements, Design, Implementation, Maintenance. In addition, certain materials are always produced (e.g. requirements document, Design, Code). These common elements can present various opportunities for better modelling a software process.

### 4.7.2 A Materials Abstraction

One method of incorporating process clarity would be to extend the RAD notation (and Romula) to encompass a Material abstraction, since materials (e.g. reports, code) are not represented formally in RADs. This allows traceability of such materials through a development process from creation to passing to another role. In addition, a CASE tool which executes or computationally analyses a RAD process model is then aware at any stage of execution what and where materials are within the process. Such minor amendments to the RAD notation and Romula can increase the power a process modelling CASE tool can provide, and promote a better focus on what is actually produced, and how, within a model. For example, such an implementation would be used as follows:

```
MATERIAL 'RequirementsDocument' TYPE 'Document'
```

When a material is passed to another role during collaboration:

```
PASSCOPY 'RequirementsDocument' TO 'Manager' TAG 'Draft'
```

And to receive a document from a role:
The *Tags* are necessary since hypothetically the same document could be passed between two roles many times and at different stages in the roles. The *Tag* system, as with thread identifiers, ensures there is no ambiguity in such situations but could be optional for situations which present no ambiguity. *Tags* can be as expressive as desired, depending on model complexity. For example, they could represent material version numbers.

Figure 4.23 shows these examples in context in their associated RAD notation.

![Figure 4.23: Example of using the Material abstraction](image)

The material typing system could then allow a modeller to inquire the location of any type of material within the executing model. The ASPEN [Doheny and Filby 1996] framework mentions four types of artifact, but for more practical purposes, the type of any material is arbitrary, ensuring that the RAD premise of 'modelling approximation' is met. This ensures that model materials are not restrained unnecessarily by type. Other such material constructs could be created, such as *Amend, Refine, Validate,* or *Verify.*

*PassCopy* implies that the passer (as opposed to the passee) of the material retains a copy, which, of course, in nearly all cases would be true.

Whilst other process modelling languages utilise an object-oriented materials mechanism, Romula's material abstraction will be unique in that it is process modelling specific. By using a real process model abstraction (instead of programming language-based object-orientation), this approach has an intuitive, approachable quality.

### 4.8 Other Applications
During the development of Romula, it became increasingly obvious that the language could be applied to modelling scenarios outside of the discipline of process modelling. At the lowest level of interpretation, Romula uses an object-to-object communication model to facilitate the representation of social processes, and by exploiting this realisation, we can exploit this communication model in other fields that require simulation. This section details two such applications to which Romula has been successfully applied. Since these two examples do not directly contribute to the subject of this thesis, and are simply included here for completeness, the reader may skip to section 4.9.

4.8.1 Modelling An Electronic Point of Sale (EPOS) System

The EPOS model discussed in this section is a simplified version of a more complex model. Discussion of the more complex version of the model is beyond the scope of this thesis.

The model presented here is based on a simple client-server architecture, with a nameserver acting as a resource locator. An executing model consists of an arbitrary number of Tills and PLUs (Price Lookup Units), and one nameserver. When a till processes a product price, it requests a free PLU from the nameserver. The nameserver locates a free PLU which is currently idle, and returns the identity of that PLU to the till. The till then requests the price for a product from that PLU. Figure 4.24 illustrates the model.

![Figure 4.24: RAD representation of the EPOS model](image)
Here, we have used a RAD to describe the behaviour of the model. RADs lend themselves well to describing component-based interactions since the modelling of components often fits within the paradigm of business rules [Henderson and Walters 1999]. As such, this RAD clearly indicates the interactions between each of the mechanistic units within the domain of the EPOS model. Figure 4.25 illustrates the Romula representation of the nameserver.

ROLE 'NameServer'
{
  EXPECTLINK FOR 'RestartService'

  DO 'Waiting to receive till request...'
  SUBINTERACT WITH 'Till' FOR 'RequestPLU' TO '$till'
  DO 'Received till request for PLU'
  INTERACT WITH 'PLU' FOR 'FindPLU' TO '$PLU'
  DO 'Found a ready PLU'

  PASSINSTANCE TO '$till' FOR 'PassPLUID' FROM '$PLU'
  DO 'Passed PLU ID to till'
  PASSINSTANCE TO '$PLU' FOR 'PassTillID' FROM '$till'
  DO 'Passed till ID to PLU'

  LINK TO 'RestartService'
}

Figure 4.25: Romula representation of the EPOS nameserver

Since the nameserver provides a recursive service, we use ExpectLink and Link to model this characteristic. Note that we have used Do to provide a means to inform the user of the progress of the model, effectively acting as a simple text output statement like printf in C, although here output is directed at the role instance window of the nameserver. The nameserver waits for any till to request a free PLU, and when this interaction is completed (storing the identity of the requesting till in $till), the nameserver interacts with an arbitrary PLU and stores its identity in $PLU. This identity is then passed to the requesting till, and the till's identity is then passed to the PLU. As will be seen later, this prevents deadlock from occurring by ensuring that the requesting till will interact with the correct PLU, and vice versa.

The Romula representation for the PLU is shown in Figure 4.26. Initially, the PLU waits for a request from the nameserver, which serves to allow the nameserver to obtain the identity of a PLU. Since the model is constrained to just one nameserver, we do not need to store its instance identity in an instance list.
The PLU then simply receives the identity of the till it is to interact with, receives the unique bar-code from that till, performs a price lookup, and sends the price to the till, and finally restarts. Although similar to the PLU definition and easily derivable from the previous definitions, the till definition is included in Figure 4.27 for completeness.

The ability to be able to model an EPOS system adequately highlights the possibility that Romula can also be applied to modelling other client-server-based architectures. The ability to model such architectures is theoretically only restricted by the inability to model communication protocols in significant detail, due to the dependence on the generalised \textit{Interact/SubInteract} mechanism to describe object-to-object communication. We can even denote request-and-reply by using an \textit{Interact} and \textit{SubInteract} command pair.
4.8.2 Modelling a State Transition Diagram

To demonstrate Romula’s modelling flexibility further, an example State Transition Diagram (STD) taken from Pressman [Pressman 1994] was represented and executed in Romula. STDs can form part of the control specification (CSPEC) for structured requirement analysis, and they model required system behaviour via a simple notation. The example taken from Pressman is given in Figure 4.28. It illustrates how a simple security system operates in terms of the states that the system can be in. At any time, the system can only be in one of the states. To traverse the state network from one state to another requires an event to be fired (Blink flag, Time out, etc.). For example, moving from the Reading User Input state to the Monitoring System Status state requires the Start/Stop Switch event to be fired.

Figure 4.28: Example of a State Transition Diagram (STD) taken from Pressman

The corresponding Romula code for two of these states is given in Figure 4.29. Each state is modelled as a role, with traversing connections represented by interactions. These interactions represent the change in control from one state to another. Since Reading User Input is the initial state of the model, the first Link To command, which jumps the first interaction command, ensures that the model does not initially have to wait for an incoming event from another state (role). If this was not done, the model would be in a deadlock, each state role constantly waiting for an incoming event to occur: each state role would be endlessly waiting for the first interaction to be completed.
When executed, the four states are represented on-screen via their respective role windows. Initially, the user can only select from the *Reading User Input* state events. By selecting one, the model changes state to either *Monitoring System Status* or *Displaying User Feedback*. A demonstration of a change of state from *Reading User Input* to *Monitoring System Status* is shown in Figures 4.30 and 4.31.
START 'ReadUserInput'
START 'MonitorSystemStatus'
START 'DisplayUserFeedback'
START 'ActingOnSensorEvent'

ROLE 'ReadUserInput'
{
    # necessary, otherwise state transition model would never start
    LINK TO 'JumpStart'
    # loop continually, accepting incoming event(s), allowing change of state,
    # and enabling this state's events to be fired
    EXPECTLINK FOR 'StateLoop'
    # accept incoming event from another state
    SUBINTERACT WITH 'ActingOnSensorEvent' 'DisplayUserFeedback' FOR 'ReadEvent'
    EXPECTLINK FOR 'JumpStart'
    DO 'Read user input'
    # enable the various events to be fired when necessary
    CONDITION 'Fire event'
    IF 'Blink flag'
        INTERACT WITH 'DisplayUserFeedback' FOR 'DisplayEvent'
    ENDIF
    IF 'Start-stop switch'
        INTERACT WITH 'MonitorSystemStatus' FOR 'ControlSystem'
    ENDIF
    ENDCOND
    LINK TO 'StateLoop'
}

ROLE 'MonitorSystemStatus'
{
    EXPECTLINK FOR 'StateLoop'
    SUBINTERACT WITH 'ReadUserInput' FOR 'ControlSystem'
    EXPECTLINK FOR 'MonitorForEvent'
    DO 'Monitor system status'
    CONDITION 'Fire event'
    IF 'No sensor event'
        LINK TO 'MonitorForEvent'
    ENDIF
    IF 'Sensor event'
        INTERACT WITH 'ActingOnSensorEvent' FOR 'SensorEvent'
    ENDIF
    IF 'Start-stop switch'
        INTERACT WITH 'DisplayUserFeedback' FOR 'DisplayEvent'
    ENDIF
    ENDCOND
    LINK TO 'StateLoop'
}

Figure 4.29: Romula representation of the State Transition Diagram given in Figure 4.28
It is a peculiarity of Romula's interaction mechanism when using `SubInteract` in a certain format which enables this behaviour to be modelled. Ordinarily, we might not have been able to model STDs in Romula because of the non-deterministic nature of events which enter a state. For example, `Displaying User Feedback` has the events `Start/Stop switch` and `Blink flag` which enable transitions to this state from `Monitoring System Status` and `Reading User Input` respectively. However, from which of these states one of these events is to be fired cannot be determined, and this creates a problem. We cannot use the following to model this behaviour:

```
CONCURRENT
TASK 'BlinkFlagEvent' :
  INTERACT WITH 'ReadUserInput' FOR 'BlinkFlag'
ENDTASK
TASK 'StartStopEvent' :
  INTERACT WITH 'MonitorSystemStatus' FOR 'StartStop'
ENDTASK
```
This is because concurrency would require both concurrent tasks to be completed before continuing, which would not occur. Only one of these interaction events would occur, and effective deadlock would result. However, Romula has had to handle various execution issues when modelling and executing RADs, and it is the way in which the interaction mechanism handles SubInteract and Interact which enables this behaviour to be successfully modelled. Essentially, when SubInteract is encountered in a role during execution, that role cannot initiate the interaction, and therefore simply waits. When a corresponding Interact is encountered, it is that role which initiates the interaction, and performs the appropriate implementation administration to conduct the interaction. This works effectively with RADs, and perfectly mirrors RAD behaviour in an ordinary context, where there are many submissive roles and one dominant role. In the real world it would be the singular dominant role which organises the interaction, since it is that role which has initiated the interaction. It is when this situation is reversed that a behavioural peculiarity emerges. With many dominant roles and a singular submissive role, it is still only each dominant role which can initiate the interaction. What effectively happens is that the singular submissive role interacts with the first dominant role which can initiate that interaction. By saying:

```
SUBINTERACT WITH 'SomeRole' 'AnotherRole' FOR 'SomeInteraction'
```

We are saying that either SomeRole or AnotherRole can interact with this role for SomeInteraction, and when one of these other roles initiates it, continue. However, we cannot pre-emptively state which of these will initiate that interaction. Hence, we can specify a type of user non-determinism which can be exploited, as demonstrated with the state transition diagram example.

### 4.8.3 Partial Ordering of Tasks

During development, some degree of validation occurred as an ongoing activity to ensure Romula remained applicable to process modelling throughout its evolution. A pseudo-realistic process model called ProcMod (see appendices B and C) emerged which was used eventually to validate Romula properly (see chapter 7). It became apparent that it was possible to model the process as a simple data flow diagram in terms of deliverables contained within this model, represented in a manner which depicted the dependencies between them (see appendix D). These dependency diagrams were shown to portray a partial ordering of the deliverables which were required throughout the project. This discovery led to an investigation into how the aforementioned partial ordering, declarative style of modelling (described in section 4.8.3.1) may be extended into a proper method. An additional activity was undertaken to determine whether such a model could be modelled in Romula.
4.8.3.1 Devising a Method of Partial Order Modelling

An alternative style of representing a process model is a declarative style which effectively allows a model to be specified in terms of a partial ordering of tasks. Although an imperative approach was adopted for Romula, this opposite style represents an alternative to modelling processes which can also effectively model materials within such processes. This style has been seen before, (see Interact/Intermediate described in section 3.3.3), but devising a separate method as discussed here was considered to be a valuable activity which placed the imperative approach adopted by Romula into perspective.

Thus, a declarative goal-based style of process programming was examined. Essentially, this entailed informally constructing a basic language set using a goal-based style with declarative linguistic properties. For an (unrealistic but functional) example, see Figure 4.32. In this example, the OUTPUTs are the goals to be completed by each process.

<table>
<thead>
<tr>
<th>PHASE ReqAnalysis</th>
<th>INPUTS: ReqReport, ReqSpec, ReqTest</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUTS:</td>
<td>ReqReport, ReqSpec, ReqTest</td>
</tr>
<tr>
<td>ROLE ReqAnalyst</td>
<td>PRODUCES ReqReport, ReqSpec</td>
</tr>
<tr>
<td>ROLE TestGroup</td>
<td>REQUIRES ReqSpec PRODUCES ReqTest</td>
</tr>
<tr>
<td>END PHASE</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PHASE Design</th>
<th>INPUTS: ReqSpec, ReqReport</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUTS: DesignDoc, DesignReport, DesignTest</td>
<td></td>
</tr>
<tr>
<td>ROLE Designer</td>
<td>REQUIRES ReqSpec, ReqReport PRODUCES DesignReport, DesignDoc</td>
</tr>
<tr>
<td>ROLE TestGroup</td>
<td>REQUIRES DesignDoc PRODUCES DesignTest</td>
</tr>
<tr>
<td>END PHASE</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.32:** An example of a declarative goal-based style of process modelling

It can be observed that this style also adopts a database worldview, since at the highest level the relationships between each of the phases are specified by project artifacts, such as ReqReport, ReqSpec, DesignDoc, etc. The completion of these deliverables allows the model to progress to the next phase. At the lower level of description, ROLES are assigned identifiers within PHASEs and REQUIRE artifacts to PRODUCE artifacts. Figure 4.33 illustrates the relationship between a PHASE and its ROLES.

When all INPUTs necessary become available from other phases, this phase can commence. Thus, we are not modelling software development phases sequentially, but concurrently; in essence, this is a producer-consumer-based flow network. Note that this approach is process-oriented as opposed to role-oriented (socio-technical); it models processes at the highest level, and not roles.
A conceptual artifact repository stores all the artifacts that are inputted to, produced by, or to be outputted from the phase. Initially, this repository stores only the inputs to the phase, but as the roles use these inputs and produce other artifacts, this repository increases, until all roles have completed their tasks and produced the OUTPUT deliverables. When this occurs, the phase is complete and the next phase can begin. It is important to note that some roles may not be able to begin until another role within the same phase has completed producing an artifact that it requires as an input. Hence, this is a method for describing partial ordering of tasks; they may be completed in any order providing the inputs necessary are available.

4.8.3.2 Implementing a Partial Order Model in Romula

To demonstrate further Romula's inherent flexibility, a successful attempt was made to model the partial ordering dependency diagram (as depicted in appendix D) in Romula. This proved to be a very straightforward task. Figure 4.34 illustrates the integration phase part of the overall data flow diagram.

The circles represent project artifacts. The dotted circles are artifacts imported from a previous phase of development that are required by this phase. Solid-lined circles represent artifacts created during this phase, whilst the more emphasised solid-lined circle represents the main artifact produced by this project phase.
Each of these artifacts then became a Romula code segment depicting how that artifact was constructed. The Romula representation for the Integrated System artifact is given in Figure 4.35. The entire Romula representation is given in appendix E.

```
ROLE 'Integrated_System'
{
    INTERACT WITH 'Validation_TestCriteria' FOR 'ValidTest-IntegSys'
    INTERACT WITH 'Modules' FOR 'Mod-IntegSys'
    INTERACT WITH 'Test_Plan' FOR 'TestPlan-IntegSys'
    DO 'Integrated System'
    INTERACT WITH 'Integration_FailuresDocument' FOR 'IntegSys-IntegFail'
    INTERACT WITH 'Integration_Report' FOR 'IntegSys-IntegRep'
}
```

**Figure 4.35:** Romula’s constructive representation of the Integrated System artifact

Each constructive artifact task is modelled much the same as this one. Each of the dependencies with other constructive artifact tasks is represented using the three `Interact` commands shown at the top of the definition. This effectively ensures that this task cannot be enacted until each of the artifacts required by this activity have been completed. The activity required to produce the artifact is then done. Following this are two `Interact` statements which model the fact that those activities are dependent on this activity to complete before they can begin.

When the model is executed, it effectively displays all the artifact construction tasks on-screen. As initial tasks are completed, other tasks which are dependent on these tasks may begin. We can easily determine from these definitions which tasks are dependent on which others. Assuming a correct representation of artifact dependency, this model cannot deadlock; the model may only be executed within the boundaries of
the defined dependencies. These dependencies dictate that the last artifacts to be produced in each phase will be produced when all prior dependent artifacts have been produced, which will always eventually be the case. Although order in the model is not specified, the model does specify completeness, in that all prior artifacts to an arbitrary artifact will be complete before that arbitrary artifact can be produced.

It is also important to note that although this representation only uses two of the Romula role commands, (Interact and Do), and does not contain the same level of modelling information as a RAD-oriented model, this representation fully models the development process described by ProcMod, which is introduced in chapter 7.

4.8.3.3 Discussion

Partial ordering offers an unconstrained approach; the software process can be completed in any manner, providing the partial ordering constraints are met. By not specifying the methods by which each phase produces deliverables and is completed, the modeller is not initially constrained by having to model unnecessary details which may be irrelevant or difficult to model.

However, this approach is not without its disadvantages. In order to provide more realistic and detailed modelling, features would have to be implemented which model the social interactions of roles; a logical extension, since we are in fact modelling roles within processes. However, roles are modelled within the scope of a phase. It would prove difficult and complicated to represent role interactions between roles in one phase and roles in another; the conceptual encapsulation the PHASE structure offers would be violated. Moreover, this type of interaction is common. For example, a Designer officially working within a Design phase may wish to discuss and clarify requirements with a Requirements Analyst before translating the requirements into a design. In a large-scale process model, a large number of these interactions would render the phase structure useless and without meaning, since the model would become predominantly socio-technical. The conflict that would occur between the process-oriented and socio-technical aspects would become ultimately very confusing.

More importantly, how would customers and project managers, who exist at a level beyond this encapsulation, be included as part of the process model? Since we are modelling role-based interactions, by extension, we should be able to model these types of role. A more fundamental problem is the possibility that a previous phase may have to be re-engineered, for example, Design. How would this eventuality be represented?

This process modelling approach provides excellent process visibility and capability at a medium-to-high abstraction level. However, it would have inherent difficulties when modelling low-level detail in concrete
models. Such process models would increasingly become ill-defined, and conceptually very complex. If such disadvantages could be overcome, then this could form the basis for a new process modelling language which would adopt an opposite perspective to Romula.

4.9 Does Romula Fulfil its Requirements?

1. Allows a capability-to-capability correspondence between the RAD notation and Romula. This would promote easier translation from RADs to Romula.

This has been demonstrated in section 4.4, where RADs have been easily translated into a corresponding Romula representation. It should be observed that from the outset, the Romula language definition was always intended to be based purely on the modelling concepts, perspective and worldview of RADs, ensuring that translating RADs into Romula always remained trivial.

2. A RAD described using Romula must be revertible to its former RAD state. If this can be achieved, then traceability exists between the two modelling approaches.

Just because it is trivial to translate RADs into Romula, it does not necessarily follow that the opposite is true. Despite this, it has been concluded that this is generally the case. Since every modelling construct in RADs has a separate, encapsulated construct in Romula, these encapsulated constructs can be reverted to a former RAD state almost as easily as translating RADs into Romula. All that is required is a working knowledge of the Romula language definition that can be gained from the RAD to Romula translation process.

However, this is not to say that all information encompassed in a Romula definition can be represented in a RAD. As discussed previously in section 4.4.6, Romula had to be extended to cope with runtime multiple role instance handling, and additional syntax was required to cope with this; the instance list mechanism introduced in section 4.4.6. Therefore, this information is lost, since there is simply no way to represent it. Even so, translating Romula into RADs is relatively straightforward, and a significant degree of traceability exists between Romula and RADs.

3. Romula is a viable process modelling method in its own right, in that models can be constructed using the language alone. This is elaborated as follows:

3.1 Romula must be syntactically concise; models created using Romula should contain a minimum of syntactic 'glue'.
3.2 Romula must be semantically clear and syntactically readable, promoting easier creation and understanding of process models.

What we have attempted to create is a language which as much as possible utilises modelling-based as opposed to programming-based concepts. However, some compromise has proved to be necessary in order to achieve this goal, semantically and syntactically.

Semantically, although we have in most cases been able to directly specify modelling-based concepts in Romula, in others, we have had to inherit programming-based concepts to ensure Romula exhibits the desired functionality. When this has proven necessary, we have been careful to ensure that a direct implementation of those programming-based concepts has not occurred; rather, that these concepts have as much as possible been adapted to fit within the modelling paradigm. An example of this is the Romula instance list. Although clearly derived from the programming variable, we have adapted its use to fit within a modelling context. As previously explained in section 4.4.6, the instance list is only used to represent 'role memory'; an instance list simply and solely contains a list of instances for referencing those role instances. Thus we have adapted its use to fit our purposes.

Syntactically, Romula is quite concise. As previously discussed in this section, much attention was given to ensuring that Romula's language definition exhibited clear traceability with RADs. To achieve this, a syntactically concise syntax was a necessity, and can be observed by the translations demonstrated in section 4.4. However, a compromise between syntactic understandability and syntactic length was required in order to ensure clear readability; Romula uses as close to natural language as possible. For example, instead of using something like $\text{InstanceList} = \text{Interact(instancelist, reason)}$ to perform an interaction which returns the instances which participated into the variable $\text{InstanceList}$, Romula uses $\text{INTERACT WITH instancelist FOR reason TO InstanceList}$ which, when read, would translate into natural language as 'This role interacts with a group of named roles for a certain reason, and the identity of those who participated is remembered as a list of names'. However, the distinction should be made between adding natural language to enhance understandability and readability, and adding syntactic 'glue' which is commonly found in programming languages. e.g. parenthesis, semi-colons, etc. It should be evident from the language definition given earlier in this chapter that Romula only contains a strictly necessary amount of syntactic 'glue'.

4.10 Summary

Chapter 3 identified certain disadvantages with existing process modelling language approaches, notably that they are low-level and programming-based. We also identified two fundamentally different approaches to arriving at an appropriate alternative: Procedural and socio-linguistic. These two approaches
are natural reversals of each other, and adopting either one of these approaches for a process modelling language can fundamentally determine the nature and required features of the language, and also to a great extent, how the actual research of a process model is conducted. In reality, constructing a process model socio-technically is easier, since determining the process from this perspective simply involves interviewing each separate role group about their activities to establish a model. Constructing a procedural perspective process model can prove to be much more difficult, as this can only be achieved by piecing together parts of that overall process from separate groups involved in that process. In effect, since procedural process knowledge is spread among multiple entities, it requires greater effort to obtain.

We have introduced a new socio-linguistic process modelling language, Romula, which attempts to overcome the disadvantages of low-level abstraction and programming-based syntax and semantics. Based on RADs, this new language has its origins firmly within the field of process modelling. This strict conceptual separation from programming languages has been demonstrated before with the requirements specification language Telos [Mylopoulos et al. 1990], where the authors state:

This language [Telos] is not a programming language. Following the example of a number of other software engineering projects, our work is based on the premise that information systems development is knowledge-intensive and that the primary responsibility of any language intended to support this task is to be able to formally represent the relevant knowledge. Accordingly, the proposed language is founded on concepts from knowledge representation.

This perfectly mirrors the rationale behind the development of Romula. Romula was developed to model socio-technical processes, and the syntax and semantics of the language have been specifically developed to achieve this task. From the outset, Romula was founded on concepts from process modelling, but when necessary, a judicious choice of programming concepts have been selected and adapted to fit the needs of the language.

If Romula had to be assigned an existing modelling style, from the classification offered by McChesney introduced in section 3.3.2, it would have to be procedural, simply because it cannot fall into any other category. This implies that it assumes an imperative style similar to that of programming languages, and to some extent, this is true. Romula is based on the RAD notation from the perspective of flow modelling; a process is represented by a sequence of tasks. In order to encompass this, an imperative style of language has been adopted, where tasks are also arranged in sequence. However, despite this, this is a natural way to view processes, and this is the perspective that was chosen for Romula.

It is this authors belief that a process modelling language designer’s first instinct when devising a suitable syntax and semantics for a process modelling language should not be to reach for the Programmer’s Guide
to Language X for inspiration, but to reach for the Process Modellers Guide to Modelling Concepts and use those concepts as the basis for those syntax and semantics. By doing so, it should be clear that models expressed in the resultant language will be more appropriate, relevant, and applicable to the field of process modelling. Romula has been designed in this fashion, and chapters 7 and 8 will demonstrate that it is capable of modelling and enacting complex processes.

It is this correspondence that enables RADs to be easily translated into Romula, and Romula models to be easily translated into RADs. This clarity also facilitates easy understanding of Romula's social process models by non-technical staff. By adopting this socio-linguistic worldview, Romula models the processes people perform, as opposed to modelling which people perform a process. However, the latter can provide a significantly different partial-ordering perspective. At a diagrammatical level it offers a clear, process relation view that can also prove beneficial by facilitating clear visualisation of how materials are related.
This chapter will specify, using a formal notation, the semantics of some aspects of the process modelling language Romula introduced in chapter 4. Section 5.1 discusses why this would be a useful activity, and section 5.2 argues in favour of a certain formal notation as the best candidate for specifying Romula. The merits of specifying Romula and an associated process modelling language in CSP together is discussed in section 5.3. Section 5.4 presents the specification of the Romula interaction and concurrency constructs, which form the core of the language, elaborating on issues and problems which were encountered during the specification process.

5.1 Why Specify Romula Semantics in a Formal Notation?

Specifying the actual linguistic semantics of Romula using a formal notation can clarify and yield greater understanding of those semantics. Up until now, we have been more concerned with modelling what a social structure is, and not how this is achieved, or what the model actually means. Although we have introduced and discussed each of the Romula language constructs previously in section 4.4, that is all we have done. We cannot adequately say what each construct does or means until we have specified those constructs formally.

By trying to understand better the semantics of Romula, we may better understand and reason about Romula models and their behaviour. A formal specification of Romula would enable formal reasoning about Romula. Various tools exist for formal notations which enable these specifications to be executed and/or analysed. This would yield a great insight into the nature of Romula models; how they operate, how they may be improved, etc.

In addition, such formal specification can provide a common baseline for evaluation of and comparison with other process modelling languages. By generalising the high-level perspectives adopted by such languages into a single formal notation, we can more easily reason about and compare their behaviour.
5.2 Why Use CSP?

As its name suggests, the notation of Communicating Sequential Processes [Hoare 1985, Hinchey and Jarvis 1995], or CSP, allows sequential processes and communications between them to be specified and promotes formal reasoning about concurrent systems, and thus lends itself rather well to modelling roles and role interactions. By representing roles as independent CSP processes, we can specify not only the semantics of how individual roles behave singularly, but how they operate collectively.

Theoretically, if Romula models could be translated into CSP, it would be possible to animate and analyse them using certain CSP tools. ProBE, (PROcess Behaviour Explorer) [Formal Systems 2000a], for example, allows the state of CSP models to be controlled and examined, the user being able to control event choices and observe how the processes evolve. This tool would assume the role of the Romula animator, although the human-computer interaction aspect would of course be different. More interesting, however, is another tool for CSP, FDR2 (Failures-Divergence Refinement) [Formal Systems 2000b]. This allows CSP models to be analysed and checked for a number of various correctness conditions, including deadlock, livelock, liveness and safety. This would therefore enable Romula models to be amenable to property checking. Using this technique for Romula definitions which model object communication, perhaps as a client-server model, may prove most beneficial.

5.3 Two Modelling Languages Linked by a Formal Notation

Work is currently being undertaken within Southampton University’s DSSE (Declarative Systems and Software Engineering) research group to specify another process modelling language, RolEnact [Phalp et al. 1998], in CSP. By having both language definitions represented in formal CSP we can then draw conclusions from the contrasts and similarities found between each approach. More interesting, however, is the possibility that by having both specified in CSP, a more generalised, high-level approach to process modelling may become apparent. Some example RolEnact, taken from a different version of the barber shop model introduced in section 4.4.6, is given in Figure 5.1.

Both RolEnact and Romula are based on Role Activity Diagrams (RADs, see section 3.2.1) [Ould 1995], but each adopts a differing perspective of how RADs operate, which results in both exhibiting different linguistic styles. Whilst Romula imperatively describes a process, with one task sequentially following another implicitly, RolEnact assumes a more explicit approach by modelling processes in terms of explicit event-driven state transitions. To simulate interactions between role instances, each role instance can alter the internal state of itself, (e.g. self.inState('cutting'), and the internal state of other role instances (e.g. aCustomer.setState('beingCut')). Each potential state transition has a guard which ensures that certain preconditions are met before the state can be altered. For example, the chooseCustomer event has two
event definitions. The Barber.chooseCustomerReady() definition ensures that chooseCustomer can only occur if the barber is ready to cut someone's hair (i.e. that barber is in the initial state), and that a customer exists and is ready for the cut (i.e. a customer exists in the initial state). Once these preconditions have been met, the chooseCustomer event can be triggered. When this occurs, the chooseCustomerDoIt() definition is activated, which essentially chooses a customer in the initial state, alters its own state to cutting, and alters the customer's state to beingCut.

```java
class Barber < Role.
Barber.addEvent('chooseCustomer).
Barber.chooseCustomerReady():=
  self.inState('initial) and
  Customer.exists('initial).
Barber.chooseCustomerDoit():=
  (aCustomer:=Customer.choose('initial);
   self.Customer:=aCustomer;
   aCustomer.Barber:=self;
   self.setState('cutting);
   aCustomer.setState('beingCut)).
Barber.addEvent('receiveMoney).
Barber.receiveMoneyReady():=
  self.inState('waitingForMoney).
Barber.receiveMoneyDoit():=
  (self.setState('initial);
   self.Customer.setState('final)).
```

Figure 5.1: Example RolEnact definition of a barber from a version of the barber shop model

Although RolEnact and Romula models represent RAD models differently and execute differently, the resultant behaviour of their models is practically identical. This is because they both manifest a socio-technical worldview, (see section 3.1), and it is only their linguistic style which differs.

5.4 Specifying the Semantics of Romula in CSP

We will show in this section how Romula interaction and concurrency may be specified in CSP, and demonstrate that this translation is a straightforward and traceable process.

The level of abstraction is an important factor when specifying CSP models. We wish to define an abstract specification, and the aim of these is to enable us to understand and to state unambiguously the way in which the system is to perform [Hinchey and Jarvis 1995], or in this case, how the system does perform. Since we already have created an implementation from a set of requirements, creating a concrete, implementationally biased model of Romula behaviour is simply not necessary, and would serve no
purpose. The aim of this chapter is to specify in formal terms the semantics of Romula, and adopting a high-level approach is the clearest, least complicated, and most understandable way to do this. It would be unnecessarily complicated to specify interaction at the level of grammar translation.

The *Do* task specification construct is basically trivial, since its only function is to inform the user of a task to perform. We simply specify *Do* tasks as CSP events.

### 5.4.1 Specification of Romula Interaction

Specifying the interaction mechanism for Romula mostly proved to be a case of providing a general solution for all possible variations of the *Interact* command:

\[
( \text{INTERACT} \mid \text{SUBINTERACT} ) \text{ WITH } ( \text{instance_list} \mid \text{role_type} ) \text{ FOR reason } [ \text{ TO instance_list } ]
\]

Firstly, the type of interaction can either be a mutual interaction (just using *Interact*), or a submissive interaction (using *SubInteract*). Secondly, the interaction target can either be a list of instances or an arbitrary role of a certain type. Lastly, an optional argument specifies whether the user wishes to capture those instances interacted with in a named instance list for later reference.

Understanding the Romula interaction mechanism requires examining the nature of the interaction, and not just its features. Figures 5.2 and 5.3 best illustrate the two types of interaction that can occur between multiple roles.

---

**Figure 5.2:** Illustration of how multiple role instances communicate in mutual (*Interact*) interactions

**Figure 5.3:** Illustration of how multiple role instances communicate in hierarchical submissive interactions, where *RoleA* is the dominant role, and the others are submissive
As introduced and discussed in section 4.4.6, role instances involved submissively in an interaction cannot initiate that interaction. Thus, in Figure 5.3, when the model is executed using the Romula animator, only $R_i$ can do this. Unfortunately, there is no way that we can specify this behaviour semantically in CSP.

In order to provide a basis for generalising the Romula interaction in CSP, a variant of the barber shop example (see section 4.4.6) was first examined. This version does not include a Reception role, and much of the complexity of its predecessor, but for the purposes of examining interactions, these refinements are not required. Figure 5.4 illustrates this simplified example as a Role Activity Diagram.

![Role Activity Diagram](image)

**Figure 5.4:** Simplified barber shop example

Firstly, the Barber chooses a Customer (i.e. he initiates the interaction). The Customer then accepts and the haircut is performed. Lastly, the Barber requests payment, the customer ends his correspondence with the Barber, and then the Barber returns to the previous state of waiting to choose a customer. In Romula, this is expressed as shown in Figure 5.5.

```plaintext
ROLE 'Barber'
{
    EXPECTLINK FOR 'WaitForCustomer'
    INTERACT WITH 'Customer' FOR 'Choose'
    SUBINTERACT WITH 'Customer' FOR 'Accept'
    INTERACT WITH 'Customer' FOR 'Payment'
    LINK TO 'WaitForCustomer'
}

ROLE 'Customer'
{
    SUBINTERACT WITH 'Barber' FOR 'Choose'
    INTERACT WITH 'Barber' FOR 'Accept'
    SUBINTERACT WITH 'Barber' FOR 'Payment'
}
```

**Figure 5.5:** Romula representation of the simplified
Specifying this model in CSP is straightforward. Since a Romula interaction requires synchronisation between involved role instances, the most logical CSP feature to represent such situations is the event, which intrinsically supports synchronisation between processes. By representing each role involved as a separate process in CSP, we can thus use events to represent communication between the Customer (C) and the Barber (B), as shown below.

\[
\text{BARBER\_SHOP} \equiv B \parallel C
\]

\[
B = \text{choose} \rightarrow \text{accept} \rightarrow \text{pay} \rightarrow B
\]

\[
C = \text{choose} \rightarrow \text{accept} \rightarrow \text{pay} \rightarrow \text{SKIP}
\]

Although both the RAD and Romula representations of this model depict who initiates various interactions, denoted by the shaded boxes in RADs and the SubInteract/Interact pair in Romula, we will not be concerned with how to specify this behaviour at this level. Due to the nature of the solution to this problem, we will cover this appropriately later in this section.

Specification at this level of detail in CSP is elegant and straightforward. Each process similarly synchronises on the choose, accept and pay events, and then the Barber B returns to the start of its definition to choose another customer, whilst the Customer C uses SKIP to indicate successful termination of its process. This specification is sufficient for one Customer and one Barber, but the problem arises when we consider multiple Barber and Customer instances. The CSP specification below could potentially have a severe problem:

\[
\text{BARBER\_SHOP} \equiv (B1 \parallel B2) \parallel (C1 \parallel C2)
\]

\[
B1 = \text{choose} \rightarrow \text{accept} \rightarrow \text{pay} \rightarrow B1
\]

\[
C1 = \text{choose} \rightarrow \text{accept} \rightarrow \text{pay} \rightarrow \text{SKIP}
\]

\[
B2 = \text{choose} \rightarrow \text{accept} \rightarrow \text{pay} \rightarrow B2
\]

\[
C2 = \text{choose} \rightarrow \text{accept} \rightarrow \text{pay} \rightarrow \text{SKIP}
\]

Suppose that we have two Barbers (B1 and B2), and two Customers (C1 and C2), and wish to model each Barber separately cutting a Customer’s hair. Unfortunately, a synchronisation problem would arise where, for example, B1 synchronises with both C1 and C2. We have specified that Barbers can only synchronise
with Customers, (using | between Barber/Barber and Customer/Customer, and || between Barber/Customer), but we have not specified that this should only occur with single Barber/Customer pairs. This is clearly incorrect.

In this situation the Romula version of this model fails also, but for a different reason. Because we have only specified the type of role to communicate with in the Romula model, we cannot be certain that the same Barber and Customer will continue the haircut appointment after the Choose interaction. The Barber could simply select any Customer instance which is waiting to interact with a Barber for a given interaction.

What is required in both cases is a method to distinguish between each single interaction pair. We have already seen in section 4.4.6 how we can distinguish between role instances in Romula using instance lists. Figure 5.6 illustrates the Romula version which uses this technique.

Role Barber

```
ROLE 'Barber'
{
  EXPECTLINK FOR 'WaitForCustomer'

  INTERACT WITH 'Customer' FOR 'Choose' TO '$customer'
  SUBINTERACT WITH '$customer' FOR 'Accept'
  INTERACT WITH '$customer' FOR 'Payment'

  LINK TO 'WaitForCustomer'
}
```

Role Customer

```
ROLE 'Customer'
{
  SUBINTERACT WITH 'Barber' FOR 'Choose' TO '$barber'
  INTERACT WITH '$barber' FOR 'Accept'
  SUBINTERACT WITH '$barber' FOR 'Payment'
}
```

**Figure 5.6:** Romula representation of the simplified barber shop example given in Figure 5.4 which uses instance lists

As with the more sophisticated barber shop model introduced in section 4.4.6, we use Romula instance lists to store the identities of the Customer and the Barber so that after the first interaction, we can ensure that the same two Barber and Customer instances communicate with each other. Hence, after the first 'role-instance-by-type selection' (i.e. selection of a role instance of a certain role type) is made between multiple role instances for the first interaction, we can use the $customer and $barber instance lists to maintain the relationship. Remember, when a certain role instance makes an interaction role-type selection, a role instance of the desired type is chosen which is waiting for the same interaction.

In CSP, we can remedy the situation in a similar manner. Firstly, we can introduce the concept of parameterisation to differentiate between individual processes (instances). By specifying, for example,
individual Barbers as $B(1), B(2) ... B(n)$ we can thus facilitate this differentiation. Secondly, we can specify role-instance-by-type selection using the external choice construct ($\square$). This signifies that a role choice is to be made by the environment of the system, in this case, between Barber and Customer instances. By capturing this choice, we can refer to the correct CSP process instance in much the same way as we would with Romula role instances. The appropriate specification for this is shown in Figure 5.7.

\[
(\big|_b B(b)) \parallel (\big|_c C(c)) \text{ where: } b \in \{1..nB\} \text{ and } c \in \{1..nC\}
\]

\[
B(b) = \square_{c \in \{1..nC\}} \langle b.c.\text{choose} \rightarrow b.c.\text{accept} \rightarrow b.c.\text{pay} \rightarrow B(b) \rangle
\]

\[
C(c) = \square_{b \in \{1..nB\}} \langle b.c.\text{choose} \rightarrow b.c.\text{accept} \rightarrow b.c.\text{pay} \rightarrow \text{SKIP} \rangle
\]

**Figure 5.7:** CSP specification of the simple Romula barber shop model given in Figure 5.6

Thus we have specified an arbitrary number of Barber (denoted by $nB$) and Customer (denoted by $nC$) process instances running in parallel with each other, and synchronising on these explicitly named events. By prefixing and hence renaming each event with $b.c.$ we have specified that synchronisation only occurs between one specific Barber and one specific Customer, which resolves the mass synchronisation problem. A choice is made by the environment as to which Customer interacts with which Barber. Since it is the Romula interpreter (i.e. the environment) which is performing this selection, (although we are not concerned with how it is done), we can assume that these choices are sensible and properly fit an executing model (i.e. that the choices made by interacting Customers and Barbers coincide with each other, and, for example, a cyclic deadlock is not encountered e.g. Customer1 waits to synchronise with Barber1, Barber1 waits to synchronise with Customer2, Customer2 waits to synchronise with Barber3, and Barber3 waits to synchronise with Customer1).

It is important to note that we can use the $\parallel$ operator to specify an arbitrary number of processes, and not be concerned with Customers and Barbers synchronising with other instances of themselves. The $b.c.$ event method ensures that each of the Customer and Barber instances only share common events with instances of the opposing type, and not their own type. This can be illustrated with a simple example. Assume Barber(1) selects Customer(3) and then specifies the $b.c.$ choose event. This would be represented as 1.3.choose, by variable substitution. Now, that Barber can only synchronise on this event with a Customer, since all the other Barbers will have events prefixed with their own unique Barber($b$) identification (i.e. other than 1), thus preventing Barber/Barber and Customer/Customer event synchronisation.

However, when we come to generalise interactions in CSP, we must realise that Romula interactions can involve any number of participating role instances. We can extend the previously explained method (e.g.
to encompass this feature, and hence allow a straightforward method of translating interactions in Romula to appropriate CSP. Consider the alternative barber shop model given in Figure 5.8. Note that the Receptionist introduced does not participate with the Barbers and the Customers in such a detailed fashion as the Receptionist in section 4.4.6, since such complexity is not required here.

We can represent this model in Romula as shown in Figure 5.9. Note that in the Barber and Customer definitions we are not specifically concerned with which Receptionist we use to conduct the Payment interaction, since we only use this arbitrarily chosen Receptionist once.

Figure 5.9: Romula representation of the barber shop model given in
Figure 5.8 which has a Receptionist introduced

We can represent this scenario in CSP as shown in Figure 5.10.

$\text{BARBER\_SHOP} =^\wedge= ((\sqsubseteq \text{B}(b)) \sqcap (\sqsubseteq \text{C}(c))) \sqcap (\sqsubseteq \text{R}(r)))$

where: $b \in \{1..nB\}$ and $c \in \{1..nC\}$ and $r \in \{1..nR\}$

$R(r) = \Box_{b \in \{1..nB\}} (\Box_{c \in \{1..nC\}} (r.b.c.\text{pay} \rightarrow R(r)))$

$B(b) = \Box_{c \in \{1..nC\}} (b.c.\text{choose} \rightarrow b.c.\text{accept} \rightarrow (\Box_{r \in \{1..nR\}} r.b.c.\text{pay} \rightarrow B(b)))$

$C(c) = \Box_{b \in \{1..nB\}} (b.c.\text{choose} \rightarrow b.c.\text{accept} \rightarrow (\Box_{r \in \{1..nR\}} r.b.c.\text{pay} \rightarrow \text{SKIP}))$

Figure 5.10: CSP specification of the Romula barber shop model given in Figure 5.9

By inserting the external choice operator (\Box) with reference to a local variable, we can effectively specify Romula's method of searching for an appropriate role instance in interactions, and thus refer to this chosen instance when required. The specification above called for a way to represent a three-way interaction between Barber, Customer and Receptionist, and this method allows us to specify such a situation.

Also interesting is the similarity of specification of role-instance-by-type selection in Romula and CSP. In Romula, Interact uses a role type to specify a role instance choice, (e.g. 'Receptionist' in Figure 5.9 means choose any Receptionist). In CSP, we similarly introduce $\Box$ with $r \in \{1..nR\}$, for example, to denote an arbitrary choice of any Receptionist. We just introduce these role-instance-by-type selections in Romula and CSP when required. In addition, the scope of local variables in Romula and CSP is the same: instance lists in Romula are created when required and remain in scope until the end of the role definition, and in CSP, role-instance-by-type selections via $\Box$ mirror this scope with correct use of parenthesis. However, this leads us on to a marked contrast between the two approaches. Romula can only specify a role-instance-by-type selection via an Interact or similar command, i.e. role instance choices are made on demand, whereas in CSP, we can make such choices at any point in the specification using $\Box$. However, for clarity we will constrain this flexibility in CSP wherever possible to provide easy traceability between Romula and associated CSP.

An interesting feature of Romula is the ability to pass and receive role instance lists, an extension to the core interaction mechanism. This enables a role instance with a specific list of other instances to pass this list to other instances. We can extend the barber shop model as shown in Figure 5.8 to adopt an identity request feature. This revised model is shown in Figure 5.11. In this model the Barber requests the identity of the next Customer from the Receptionist. Both the Customer and the Barber check in at reception, the Barber obtains the identity of the Customer, and the model continues as before. Note that unlike the
previous barber shop model, since it is the Barber who selects the Customer, and therefore initiates the Accept interaction, we have mirrored this change in the above model.

![Diagram of the modified barber shop model](image)

**Figure 5.11**: The alternative barber shop model given in Figure 5.8 with an added 'checking-in' feature

In Romula we can use the PassInstance and GetInstance features to mirror this behaviour. This is shown in Figure 5.12.
So how can we model this identity passing between role instances in CSP? Fortunately, CSP has a well implemented communication mechanism which can appropriately model this behaviour, and is provided in the form of specialised events. We can use \texttt{event!sendValue} to send and \texttt{event?receiveValue} to receive, where the communication synchronises on \texttt{event}. Therefore, in CSP we can specify the model given in Figure 5.12 as shown in Figure 5.13.

\[
\text{BARBER\_SHOP =}^\bowtie ( (\parallel_{b \in \{1..nB\}} B(b)) \parallel (\parallel_{c \in \{1..nC\}} C(c)) \parallel (\parallel_{r \in \{1..nR\}} R(r)))
\]

where: \( b \in \{1..nB\} \) and \( c \in \{1..nC\} \) and \( r \in \{1..nR\} \)

\[
R(r) = \square_{c \in \{1..nC\}} (\text{r.c.checkin} \rightarrow (\square_{b \in \{1..nB\}} \text{r.b.checkin} \rightarrow \text{r.b.choose!c} \rightarrow \text{r.b.c.pay} \rightarrow R(r)))
\]

\[
B(b) = \square_{r \in \{1..nR\}} (\text{r.b.checkin} \rightarrow (\text{r.b.choose?c} \rightarrow \text{b.c.accept} \rightarrow \text{r.b.c.pay} \rightarrow B(b)))
\]

\[
C(c) = \square_{r \in \{1..nR\}} (\text{r.c.checkin} \rightarrow (\square_{b \in \{1..nB\}} \text{b.c.accept} \rightarrow \text{r.b.c.pay} \rightarrow SKIP))
\]
Therefore, whenever we require an instance to be passed between role instances, we simply use the ! and ? convention to provide this functionality. However, there is a restriction with regard to this method. We cannot pass lists of role instances between each other, as we would in Romula, simply because they cannot be used. We cannot write set.event to denote event interaction between all role instances in set since CSP does not support this format. Therefore, we must restrict instance list passing to singular role instances.

5.4.2 Specification of Romula Concurrency

Figure 5.14 illustrates how we may introduce concurrency into the initial barber shop model given in Figure 5.11 in section 5.4.1. In this revised version, we have Barber and Customer 'checking in' with the Receptionist concurrently. This is an improvement over the previous barber shop model shown in Figure 5.11 in section 5.4.1, since we are allowing both the Barber and Customer to check in with the Receptionist asynchronously, and then perform other tasks until the main Barber/Customer interactions can be initiated.

Figure 5.14: Introducing concurrency into the barber shop model
The other difference is that the *Barber* chooses the *Customer* through the *Receptionist* before selecting the *Customer*. Thus, this better represents the real-life situation, as either a *Barber* or *Customer* has to wait for an instance of each other to be available before commencing the haircut, which can happen at any time. They are not constrained by order as in the model given in Figure 5.10, where the *Customer* had to check in before the *Barber*; in this model they can check in independently. During this waiting time, the *Customer* can read a magazine, whilst the *Barber* can clean and prepare the cutting equipment to perform the next haircut. This situation can be modelled in Romula as shown in Figure 5.15.

Note that because the *Barber* and *Customer* initially interact with the *Receptionist* on a concurrent level, both the *Barber* and the *Customer* would have 'Receptionist.WaitForBarber' and 'Receptionist.WaitForCustomer' respectively within their '$receptionist' instance lists. We must thus deprecate these instance lists to just 'Receptionist' to ensure that these roles interact correctly with the 'main' thread of the *Receptionist* when required later, using the convention '$instance.MAIN'. Figure 5.16 illustrates the CSP specification of the more complex Romula barber shop model given in Figure 5.15.
The **Receptionist** is first required to wait for the **Customer** and the **Barber** to 'check-in', which they will do at arbitrary times. We specify these concurrent tasks in a CSP role process using the || operator, and in this situation we use SKIP in both of these tasks to specify synchronisation between these two concurrent processes, reflecting the synchronisation which must occur in Romula concurrency before the rest of the process model can continue. Specifying this continuation in CSP is also straightforward; we specify the rest of the process after the ; operator, which denotes that after both of these processes have synchronised and completed, we can continue with the rest of the specification. Note that we could not have simply
specified \((ConcurrentProcess1 \parallel ConcurrentProcess2) \rightarrow Event\) since this would constitute an invalid specification.

\[\text{BARBER\_SHOP} =^= ((||_{b \in \{1..nB\}} B(b)) \parallel (||_{c \in \{1..nC\}} C(c))) \parallel (||_{r \in \{1..nR\}} R(r)))\]

where: \(b \in \{1..nB\}\) and \(c \in \{1..nC\}\) and \(r \in \{1..nR\}\)

\[R(r) = \square_{b \in \{1..nB\}} (\square_{c \in \{1..nC\}} ((r.b\text{-checkin} \rightarrow SKIP) \parallel (r.c\text{-checkin} \rightarrow SKIP)); (r.b\text{-choose}!c \rightarrow r.b.c\text{-pay} \rightarrow R(r))))\]

\[B(b) = \square_{r \in \{1..nR\}} (r.b\text{-checkin} \rightarrow b\text{-prepare} \rightarrow r.b\text{-choose}?c \rightarrow b.c\text{-select} \rightarrow b.c\text{-accept} \rightarrow r.b.c\text{-pay} \rightarrow B(b))\]

\[C(c) = \square_{r \in \{1..nR\}} (r.c\text{-checkin} \rightarrow c\text{-read} \rightarrow (\square_{b \in \{1..nB\}} b.c\text{-select} \rightarrow b.c\text{-accept} \rightarrow r.b.c\text{-pay} \rightarrow SKIP))\]

\textbf{Figure 5.16:} CSP specification of the more complex Romula barber shop model given in Figure 5.15

Thus, specification of Romula concurrency is quite straightforward. In situations where more than two concurrent processes are required, we can simply specify additional CSP concurrency using the \((ConcurrentProcess1 \parallel ... \parallel ConcurrentProcessN)\) form for \(N\) concurrent Romula tasks.

As far as the model is concerned, we could further extend it to make it more realistic. Up until this point, we have only considered single pairs of \textit{Barbers} and \textit{Customers}, with one \textit{Receptionist}. This is fine, until you consider the possibility of multiple pairs of \textit{Barbers} and \textit{Customers}. Only one pair of these can operate with the \textit{Receptionist} at any given time, since after a \textit{Barber} and \textit{Customer} pair has checked-in asynchronously the \textit{Receptionist} cannot check any others in until after the payment interaction has concluded with that same \textit{Barber/Customer} pair. We could improve this situation by simply specifying multiple \textit{Receptionists} to cater for more \textit{Barber/Customer} pairs, but a better solution would be to specify the payment interaction in parallel with the checking-in interactions, thereby ensuring that \textit{Barber/Customer} pairs may check-in whilst other pairs are conducting payment.
5.5 Summary

We have demonstrated in this chapter how the core interaction and concurrency semantics of Romula can be specified using a formal specification notation, in this case, CSP. This process has enabled us to reason about the language features offered by Romula from a formal perspective, and as a consequence, has allowed us to be able to reason about Romula models formally. It is now possible to translate Romula into CSP and then execute the CSP on a CSP animation tool. We have achieved this by using a CSP stepper based on the Enact programming language [Enact 2000], which is the base language for RolEnact [Phalp et al. 1998]. This has enabled us to validate our CSP specifications, and observe their behaviour. However, it would also prove most interesting to see how commercial implementations of CSP animation and analysis handle these Romula-based models, such as ProBE [Formal Systems 2000a], and better still, FDR2 [Formal Systems 2000b]. Analysis of Romula-based CSP could yield interesting information concerning the properties of Romula models, allowing us to check for such properties as deadlock, livelock, and reachability. This could prove most beneficial to applications outside the field of process modelling. Models such as the EPOS system modelled in Romula in section 4.8.1, for example, may prove good candidates for such analysis.

CSP has proven itself to be a good choice for expressing the semantics of Romula for the following reasons:

- The process of translating Romula into CSP, even at high levels of model complexity, has been demonstrated to be straightforward.
- We can easily specify the selection mechanism of Romula's role interactions using the CSP external choice operator $\exists$.
- We can easily and clearly specify Romula's concurrency construct using $(ConcurrentProcess1 || ... || ConcurrentProcessN) ;...$
- The event mechanism in CSP perfectly specifies the synchronisation behaviour which occurs between roles in Romula when they interact.
- The CSP communication system can be used to specify the instance list passing behaviour between roles.
- Clear traceability exists between a model in Romula and the corresponding CSP specification. We can also adopt a CSP presentation 'style' to reflect the structure of Romula to enhance this traceability.

However, We cannot completely specify instance list passing, since we can only specify the passing of singular instances. Despite this restriction, for most scenarios this would be enough.
The high level of traceability between Romula and CSP is considerable, but perhaps not totally surprising. We have based Romula on the RAD notation, and the RAD notation is based upon a state transition system. State does exist in Romula, although implicit, and this implicit state is made explicit by translation into CSP. RolEnact models, on the other hand, exploit the RAD state transition system by integrating it into a series of event-driven state transitions.

To illustrate an added advantage of translating socio-technical process modelling languages into CSP, let us consider a large process which is to be represented by two separate process models. One of these models is to be modelled in RolEnact, and the other in Romula. We cannot animate the entire model since these two language platforms are directly incompatible. However, by translating them both into CSP, it is conceivable that we could execute the resultant integrated model as one model. The advantage of this approach is that whilst the Romula modeller and RolEnact modeller continue to update their models in their own languages, it would be possible at convenient points to translate them into CSP, perhaps using software-based automated translation tools, and then examine the entire model's behaviour by executing that CSP. A more interesting prospect, however, is that we could run the entire model through a CSP analyser and check various properties of the amalgamated model. We would therefore effectively be checking model properties across two process modelling platforms.

To close this chapter, a quote from Augusta Ada Lovelace, referenced by [Hinchey and Jarvis 1995], sums up both Romula's intended application domain, and that of CSP:

There are frequently several distinct sets of effects going on simultaneously; all in a manner independent of each other, and yet to a greater or less degree exercising a mutual influence.

The 'distinct sets' in Romula are those of roles, which model the activities of people, each having their own agenda, yet influencing by interaction the activities of others. The 'distinct sets' in CSP, of course, would ordinarily refer to concurrent processes within computers. However, in this chapter we have essentially used CSP as a tool to describe social processes, which seems to reflect on the unexpected application of programs to model such processes which has become recognised over recent years.
In order to validate Romula as a process modelling tool, we need to demonstrate that it can suitably model and execute processes of substantial complexity and scope. This chapter provides some relevant background for the next two chapters, which each demonstrate via example how Romula may be used to model and organise such large-scale processes. With the recent increase in focus towards requirements engineering, chapter 7 introduces a model which incorporates some requirements gathering and validation features, whilst chapter 8 introduces a process framework which attempts to demonstrate how requirements and the associated process which fulfils those requirements may be represented within a single conceptual definition. Thus, this chapter will introduce and discuss how requirements are handled within a software project. Section 6.1 serves as an introduction to the requirements phase, and section 6.2 offers some motivations for investigating the requirements phase. Sections 6.3 and 6.4 discuss the nature of functional and non-functional requirements respectively, and section 6.5 introduces two existing CASE tools which illustrate how focus on requirements can be maintained throughout the software development process.

### 6.1 Introduction to the Requirements Phase

Essentially, the development of software is requirements-oriented. Requirement analysis is the key phase in the development process which defines the products' functionality (functional requirements) and operating characteristics (non-functional requirements). Without well-defined requirements, an ill-defined product results which is unable to integrate usefully and successfully into its operational domain. Without a consistent, well defined set of customer's requirements, how can the final product be effectively validated with respect to customer satisfaction?

In order to produce well-defined requirements, it is imperative that the difference between and the importance of functional requirements \((FRs)\), and non-functional requirements \((NFRs)\) is realised. Although this thesis later deals with FRs (chapter 8), it is important when dealing with this type of requirement to place them conceptually in context with NFRs. Sections 6.3 and 6.4 discuss these issues of understanding, representing, and classifying NFRs.
6.2 Why Study Requirements?

The result of the software engineering process, the software product, is determined by the defined product requirements. But to a greatly varying degree, requirements remain poorly supported [Borgida 1985], poorly represented [Ramesh and Dhar 1992], inconcise [Chung et al. 1991b] and badly maintained [Chung et al. 1995]. A requirements engineering process which has been integrated into the software development lifecycle must maintain high priority if a quality product is to result.

The lack of a definitive framework which stresses the importance of requirements analysis, and guides project teams to produce quality requirements' specifications at earlier project lifecycle stages, has become increasingly evident. The Gersick 'shutting down' phenomenon of prematurely ending requirements analysis based on project timing as opposed to gaining a full understanding of requirements appears prevalent. Walz et al's [Walz et al. 1993] study of the activities of an actual software design team highlights this phenomenon, and concludes: 'requirements determination did not end cleanly, but was a lengthy process that seemed to "shut down" based more on project timing than on achieving a full understanding of requirements'. Surely if full understanding of requirements is not achieved, how can the design team fulfil the customer's expectation of the software? The resulting shift in the priority of obtaining various information types (knowledge, requirements, design approaches) in the development process is shown in Figure 6.1. Closeness to the centre of each circle represents those information types with higher requisitional priority. The raised inner circles represent the information type with the highest priority at that stage of the development process.

![Figure 6.1: Shifts in project phase emphasis during the development process](image)

The diagram illustrates the focus on each of the activities carried out up until the design phase. As can be seen, requirements maintains high focus throughout the initial processes of software development. Indeed, over 75% of the time devoted to the design phase of the observed project was spent in learning. In
practice, however, Walz observed that much of this information given by the customer was simply not captured, and this led to time consuming reconstruction at later stages of development.

In addition, Walz highlights the lack of sufficient, and proficient, documentation. The apparent lack of requirements clarification led to inconsistent and incomplete requirements specification. This also led to new team members being unclear on what was achieved so far, creating many misunderstandings and much lack of knowledge. What must be made clear about these findings is that they are not statistically representative of all software development teams and their activities; the study focused on one single development team, and Walz states that a broader range of empirical research on software design teams is necessary. Nevertheless, these observations provide a valuable insight into real-world software design practice, and so gives us a real basis for evaluating applicative research.

6.3 The Nature of Functional Requirements

With reference to Lehman's Heisenberg-like uncertainty, as a system progresses through its lifecycle, more needs (requirements) are discovered which were not initially identified; the system therefore constantly evolves, sometimes very quickly. Clearly, new and old requirements need to be handled proficiently, and related, so a consistent and unconflicting new system is developed.

Lehman details the benefits of CASE, asserting that the handling of assumptions is the key to satisfactory system application. It is clear that CASE holds great potential to aid the requirements analyst. Indeed, Lehman later asserts that it is during the analysis stage, while 'requirements [are] analysed and developed, that relationships are recognised or established and basic assumptions made'. Introducing formalism into requirements analysis and providing tool support, he argues, 'could yield major benefit in terms of greater precision in implementation, increased effectiveness, improved documentation, more responsive adaption and less uncertainty about system properties'.

In agreement with Lehman, Walz [Walz et al. 1993] states that for software to be developed on a predictable scale requirements must be established and maintained with reasonable stability throughout the development cycle. The consequence of not providing such requirements consistency, he claims, can lead to unordered design, implementation, and testing. In addition, since requirements are not consistent, an ineffective quality plan can result.

A framework which attempts to help maintain a high level of focus on requirements is given in chapter 8.
6.3.1 Representing Functional Requirements

Traditionally, functional requirements are represented hierarchically [Henderson 1999], allowing top-level requirements to be systematically refined into more specific requirements. Figure 6.2 displays an example requirements hierarchy. This approach allows a clear definition and representation of requirements as they are recognised and refined.

Software project requirements are not mandatorily software-related. For example a certain type of user manual is a requirement of any software project.

A more practical alternative to a simple requirements hierarchy is a hierarchy of requirement lists [Henderson 1999]. Initially, a top-level list of requirements is created, which would realistically be contained in the initial Project Plan. These initial requirements are examined and elaborated, and refined into sub-requirement lists. Each sub-requirement list may constitute a set of requirements for a design team internal to the developing company, or even a workload of requirements to pass to a sub-contracting company. Each sub-requirement list is then analysed by the new group responsible for fulfilling those requirements, and the list is refined into further sub-requirement lists. Those requirements are delegated, and so on and so forth. Each requirement list describes a desired component, and as each list of requirements is finished, a completed component is returned to the entity that requested that component.

6.3.2 Fulfilling Functional Requirements: The Completion Factor
At the highest level, the process of fulfilling a list of requirements is simple: the requirements are analysed, component(s) are engineered to meet those requirements, and those components are finally validated to ensure they fulfil the requirements.

The possibility exists that one, or more, requirements may remain unfulfilled, perhaps because of time constraints or a failed validation test. So what happens if a sub-requirements list has perhaps only been 90% completed by a sub-contractor? In this instance, it must be decided whether or not to accept such an incomplete component, by implementing methods of requirements list representation, measuring requirement list completeness, and a policy of acceptance based on completeness. An example of a possible requirements list representation is given in Figure 6.3.

![Figure 6.3: Example of a requirements list representation](image)

Initially, a list of requirements with a blank completion column is presented to a sub-group manager. As the sub-group progresses with the workload, the requirements are marked off on the list as they are completed. The complexity of completing this list is not to be underestimated; that sub-group may sub-contract to another group to complete some of those requirements, or perhaps refine one of those requirements into another set of requirements to sub-contract. Figure 6.4 illustrates how a requirements list hierarchy could be represented, and demonstrates the possible complexity of the problem.
A group (internal department, or sub-contractor) responsible for a requirement list initially refines those requirements to further their understanding of the requirements. They may choose to do one or a mix of the following with their assigned requirement list:

1. **Engineer Requirements**: Simply use the resources available to the group to complete requirements. They may not be able to provide an adequate solution for all requirements, perhaps due to time constraints.

2. **Sub-Contract Requirements**: One or more of the requirements in the list are passed to sub-contractors, possibly specialists in completing certain types of requirement. The sub-contractors may also be unable to provide an adequate solution for all their assigned requirements.

Each of the groups involved in this hierarchy is therefore responsible for completing the requirements assigned to them.

This complex, but realistic, model of viewing the handling of requirements raises the issue of completeness. Completeness in this case is defined to be the degree to which all the requirements have been successfully validated. What level of requirements completeness is necessary for a successful component return? 70%? 99%? This illustrates the necessity of providing a method of measuring completeness based on the number of requirements, importance of each requirement, and whether each requirement has been successfully validated.
validated. Chapter 8 introduces a framework based on this requirements model which attempts to reduce, or at least highlight, such risk by providing knowledge of the current state of a project.

6.4 The Nature of Non-Functional Requirements (NFRs)

It is important to understand the fundamental differences between functional and non-functional requirements. Otherwise, implicit inclusion of NFRs within the scope of FRs can result. Although the term 'non-functional requirement' misleadingly implies a requirement which is simply not functional, much literature [Chung 1991a, Chung et al. 1991b, Chung et al. 1995, Mylopoulos et al. 1992] is devoted to defining the role, relevance and importance of NFRs. NFRs basically represent the quality attributes of the software; the required level of software quality. Unfortunately, NFRs are all too often defined implicitly and/or poorly represented.

As mentioned in section 2.2, basic assumptions concerning the software to be produced are made during the requirements analysis phase. Many such assumptions are those of the non-functional variety; performance, usability, reliability, etc. By attempting to eliminate these assumptions by defining them explicitly as non-functional requirements, a far more effective requirements specification can result. Additionally, the NFRs, which address quality issues, can form a major part of the product's quality plan, which can be used as a 'quality measure' for product validation purposes later in the development process. Importantly, clear definition of the boundaries between functional and non-functional requirements can only benefit the overall understanding of the requirements themselves and the real-world domain to which they refer. The next two sections deal with the representation and classification of NFRs respectively.

6.4.1 Representing NFRs

Chung [Chung 1991a], a prominent figure in NFR research, details a comprehensive and involved NFR representation and utilisation methodology. He proposes a goal-oriented approach, presenting two methods: goal decomposition (GDMs) and goal satisficing (GSMs), usage of these results in methods for goal composition to assess design quality. Satisficing is a term that indicates the software produced will fulfil its requirements within acceptable limits, since there is no formal definition of when a software system satisfies a set of non-functional requirements. NFRs can be assessed in terms of co-operation or conflicts towards a design, and he focuses on the NFR ‘accuracy’ as an example throughout the paper to illustrate his methodologies.

Chung argues that NFRs are rarely completely fulfilled by design decisions, and that more realistically such decisions contribute either positively or negatively towards a given NFR. He also observes that functional
requirements (FRs) typically have only localised effects on design decisions, whilst NFRs are more global in nature: they effect many design components. This appears intuitively to be correct, and this observation does indicate that NFRs must be clearly and appropriately expressed and understood if they are to be fulfilled by all affected design components.

Chung’s work is revised and updated in two later papers [Mylopoulos et al. 1991, Chung et al. 1995]. In collaboration with Chung and Nixon, Mylopoulos [Mylopoulos et al. 1991] focuses on a framework which provides increased traceability from design to requirements. Many refinements and extensions exist over the framework given in [Chung 1991a], notably with the addition of argumentation goals and argumentation methods, which indicate evidence/counter-evidence, in terms of arguments, for the satisficing of a goal. Additional formal notation and methods are also given to deal with correlation rules which allow the designer to select those satisficing goals which best meet a set of given NFR goals. However, the methods provided, although very comprehensive and detailed, are best described as ‘intensely mathematically oriented’, and appear to deal with abstract concepts (i.e. NFRs) in a very low-level manner. Perhaps this is why that although the paper demonstrates how to use the framework with accuracy and performance NFRs, in conclusion, the paper states that ‘formal semantic treatment’ would have to be done individually for every type of requirement, and is therefore a long-term project. In short, the formal methods given provide a basis for further research, but have limited practicality in the real world due to their specific nature of implementation.

Perhaps Chung and Nixon, realising this, decided to provide a more overall approach to dealing with NFRs (in addition to a separate, more involved version) since their later paper [Chung et al. 1995] gives a more realistic overview of how their general principles can be applied. They focus on a change of banking policies at Barclays Bank, as opposed to a more hypothetical and theoretical examination of accuracy and performance requirements as in the previous papers. In addition to Lehman and Watts, the authors have identified the importance of changing and conflicting requirements in software development, to which they reference an empirical study. The very nature of this version of the paper appears to be more ‘approachable’ and more readily understandable by others outside the area of NFR research, since a more realistic scenario-oriented presentation has been adopted.

### 6.4.2 Classification of NFRs

A report published by the Rome Air Development Center (RADC) [Bowen 1985] classifies NFRs into consumer-oriented (software quality factors), and technically-oriented (software quality criteria) attributes. White and Edwards [White and Edwards 1995], in attempting to produce an all-encompassing requirements taxonomy have produced RE-Views, which does appear to have greater breadth than other comparable taxonomies in addressing non-functional and operational requirements. However, it could be
argued that there is no complete taxonomy of NFRs for every possible project, because an all-encompassing taxonomy would place bias in areas which might not be of direct relevance to certain software projects (e.g. survivability). Mylopoulos et al [Mylopoulos et al. 1992] adds weight to this argument, stressing that although a referenced table displaying (consumer) quality concerns (NFRs) applies to all software systems, additional requirements may apply for special classes of software. This suggests that a singular, common base taxonomy could form a starting point in producing more comprehensive taxonomies tailored to individual projects.

The BSi document on software categorisation [BS ISO/IEC TR 12182, 1998], ISO/IEC 9126 [ISO/IEC 9126, 1991], and indeed the RADC classification are very similar in structure. For example, all three of these proposed classifications adopt a two-tier approach (e.g. Reliability, Efficiency, Integrity are main classifications, and Accuracy, Anomaly Management, and Simplicity are sub-classifications of Reliability, as given by the RADC paper). Such commonality should be exploited, and perhaps support for a common subset of all classifications could be provided in an IPSE or other such tool [ISO/IEC 9126, 1991]. In addition, the RADC taxonomy provides table of Software Quality Factor Interrelationships, showing positive and negative effects of various NFRs in relation to each other, and this approach has been seen before [Mylopoulos et al. 1992]. The table given in the RADC document addressing consumer quality 'concerns' relates each of these concerns to NFRs which affect that user concern. These could also be provided as support in an IPSE or similar tool.

6.5 CASE as a Means of Maintaining Focus on Requirements

This section introduces CASE (Computer Aided Software Engineering) as a means of assisting the software development process. It has already been emphasised that requirements should maintain a high level of focus throughout the software process, and there are two approaches which in part attempt to do this. These are the DAIDA and ASPEN frameworks, and each is introduced in sections 6.5.2 and 6.5.3 respectively. These approaches will be covered in a significant level of detail, since they also exhibit other features that are also relevant to process modelling and requirements analysis. In addition, see chapter 7 which validates Romula by applying it to modelling an example requirements-focused process model, and chapter 8, which introduces an organisational process framework also created for the validation of Romula which combines requirements with their associated fulfilment process.

6.5.1 Introduction to CASE
Walz et al [Walz et al. 1993] advocate the use of CASE as a tool for team knowledge acquisition throughout a software project. They state that 'New computer-based tools are needed to easily and unobtrusively capture this process-based information'. Additionally, that such tools are able to capture design rationale and requirements including scenarios of use as supplied by customers and designers. In short, improving overall team awareness of a software project, helping to prevent misunderstandings and ensuring all team members (including new arrivals) are kept up to date on the latest developments.

Lehman [Lehman 1991] also strongly advocates CASE, explaining its role(s), and effectiveness. As mentioned previously, he argues that introducing formalism into requirements analysis and providing tool support 'could yield major benefit in terms of greater precision in implementation, increased effectiveness, improved documentation, more responsive adaption and less uncertainty about system properties'. He stresses the potential importance of CASE, arguing that 'the demand for software has outpaced the availability of adequate science and technology to produce it', and that proper program evaluation (to achieve satisfaction) implies the use of method, formality and mechanical support (i.e. CASE). Much detail is given concerning uncertainty; and three types of uncertainty are put forward. Two of these bear much relevance to this thesis: Process uncertainty and Heisenburg-like uncertainty. The separate disciplines of process modelling and requirements analysis attempt to alleviate the risks associated with these uncertainties respectively. An overview of these types of uncertainty addressed by Lehman can be seen in section 2.2.

Christer Fernstrom et al [Fernstrom et al. 1992], of the Eureka Software Factory (ESF) project, focus on CASE, its role, tools and IPSEs within a software factory analogy. They stress that this analogy can only be applied to the goal of industrial production, not its implementation, because little or no traditional production exists in software development. A software factory analogy focuses on co-ordinating information between producers and consumers: this enables the right person to have the right information at the right time. They cover information logistics, stating there are three levels (organisational, team, and individual) which any environment must support. More interesting, however, is their assertion that knowledge and experience are most often collected only informally, which confirms a widely stated belief noted in previous papers. This paper provides an abstract model for CASE applications, ensuring they are not 'isolated' solutions, but fully integratable with other CASE tools. This is achieved in part by providing a simple classification framework (see Figure 6.5).

<table>
<thead>
<tr>
<th>Product</th>
<th>Supported Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Operation</td>
</tr>
<tr>
<td>Tool</td>
<td>Task</td>
</tr>
<tr>
<td>Tool set</td>
<td>Role</td>
</tr>
<tr>
<td>Environment</td>
<td>Production process</td>
</tr>
</tbody>
</table>

**Figure 6.5: CASE Product Types**
This method classifies CASE tools by the individual part of the software process it supports. Another method of classifying CASE tools is given later, in section 6.5.4 (Applying the Maturity Model to CASE?).

6.5.2 DAIDA

DAIDA [Jarke 1991a] does much to emphasise the awareness of requirements throughout development. Beginning life in the early 1980's, it developed over the years until a successful prototype was demonstrated and the project was completed in mid-1990, although several related activities continued until 1991.

The DAIDA framework provides a multi-layered conceptual model of software:

1. **Requirements Model**: Consists of captured application perspectives which represent the different system roles in the knowledge representation language Telos [Jeusfeld 1991].


3. **Programming**: Subsystems implemented in database programming language DBPL [Schmidt 1991].

It must be noted that the overall model does not view the development process as a stage-to-stage framework, but as a hierarchical framework, with requirements at the highest level.

The DAIDA CoNeX prototype software allows a project to be represented initially only as a requirements model, ensuring design issues are not imposed prematurely upon project requirements. By following such a strict protocol, superceding design can only be conceptually represented in these models in strict relation to the requirements model, ensuring design rationales can be suitably traced back to their corresponding requirements. It is this assuming property that maintains the focus on requirements until design is complete, preventing development teams inefficiently spending perhaps 75% of the design phase still learning and understanding the requirements phase [Walz 1993].

Knowledge-based mapping assistants exist in the prototype for transitions to the next conceptual layer, allowing derivations of formally verified database programs from conceptual designs. An example given in the text is presented in Figure 6.6.
The DAIDA architecture requires three different viewpoints related by development decisions. Represented as formal objects of a knowledge base, these are *EmpPerPro_REQ*, the requirement analysis object, *EmpPerPro_Des*, the system perspective, and *EmpPerPro_Imp*, the database program implementation. Mappings exist between each conceptual layer which represent decisions taken to achieve the next layer.
A zooming operation allows the user to examine the semantic descriptions of objects. See Figure 6.7 for a cut-screen example. The requirements model consists of four entity classes (Person, Employee, Company, Project) and three transactions (hireEmployee, hireEmployeeforProject (hireEfP), fireEmp). It is important to note that the diagram does not exhibit a one-to-one correspondence between the requirements data classes and the design data classes. The actual mapping between the two is specified in the InitialReqMap definition. Where designers merge separate requirement data classes into one design class, dependency links are shown on-screen. (The actual prototype shows named dependency links, although the more detailed ones are missed out here for clarity). For example, designers have merged Person and Employee into a single EmplPers design data class, so each requirements model activity (hireEmp, hireEfP) is mapped to an individual TDL transaction. Thus, users can trace design history details for reuse or correction.

The CoNeX implementation is programmed in X-Windows using the GraFlc graphical data and window management system [Rouge 1991], and Figure 6.8 shows the simple yet practical control panels. The top panel allows access to the program's functionality, whilst the lower panel displays the main graphics.
window. Option exists to alter the graphical scale factor of the graphics window, redisplay (supposedly after altering the scale factor), and update the configuration (supposedly after altering the graphics in the main window). The message area displays information concerning a selected object.

The graphics in the main window are handled using a direct manipulation interface method, allowing the user full visualisation and control over the displayed data objects. This increasingly popular method of data visualisation and manipulation provides many advantages, as highlighted by Shneiderman is A. Dix et al [Dix et al. 1993]:

1. Visibility of the objects of interest.
2. Incremental action at the interface with rapid feedback on all actions.
3. Reversibility of all actions, so that users are encouraged to explore without severe penalties.
4. Syntactic correctness of all actions, so that every user action is a legal operation.
5. Replacement of complex command languages with actions to manipulate directly the visible objects.

It is clear that direct manipulation is a suitable method employed by the CoNeX prototype, although since it is only a prototype, it probably does not meet all of the above stated requirements of a proper direct manipulation interface, although usage of the system would be required to determine this. However, it is clear that '3' is at least partially catered for, since correct changes can be updated into the database when required, although supposedly incorrect changes cannot actually be reversed. Requirement '5' is
interesting, since the prototype uses the languages Telos and TaxisDL as functional support for the graphical interface, and forms an integral part of the system.

6.5.3 ASPEN

The ASPEN [Doheny and Filby 1996] process modelling framework was introduced at the Software Quality Management conference in 1996. It identifies and formalises project information models, concepts, and representation, project activities and agents, and project artifacts:

- **Information Models**: The framework provides four types of information model, which bear much similarity to the DAIDA method; Real-World, Requirements Model, Computation Model, and Implementation Model. Figure 6.9 clarifies the simple relationship.

![Figure 6.9: ASPEN Information Models](image)

- **Information Concepts**: The content of each of these models contains an abstracted description of that part of the application domain of interest. Information concerning each model is decomposed into Rationale/Goal (Why), Entities (What), Processing (How), Agents (Who), Location (Where), and Dynamics (When).

- **Information Representation**: How the information is represented (e.g. textual, graphical, audio, etc.) and media onto which it is stored. Interestingly, the degree of formality of information representation is also mentioned; being informal, structured and formal.

- **Activity and Agent Models**: Project processes and involved roles are represented separately as Activity Models and Agent Models. Project Activities utilise project artifacts to produce project artifacts. Agents perform these activities, and can be real personnel, or a tool. A declarative balance has been created between process and agent. Instead of defining a process model as activities within roles, or vice versa, a process framework is constructed with conceptual separation from each type of model, with relationships defined between the two, and even between instances of themselves.
A Project Activity Model is given in Figure 6.10.

As with DAIDA, the Requirements Model exists as a separate and prominent part of the development process. The framework stresses that the Requirements Model, whilst describing software functions, should not contain any information that unnecessarily constrains possible designs and implementations.

![ASPEN Project Activity Model](image)

**Figure 6.10:** ASPEN Project Activity Model

The ASPEN framework also provides a process assessment method which focuses upon conformance to standards and process effectiveness. Assessing the effectiveness of a development process is achieved by four basic steps:

- **Strategy:** Considers overall approach to technical development, verification & validation or risk analysis. A development strategy is concerned with the lifecycle and maturity of approach, user involvement in the process, existing strategy re-use, etc.

- **Completeness:** Completeness of the contents of the project artifacts. Completeness of verification & validation and risk analysis relates to the level of testing and
analysis of the various technical artifacts produced by the project. e.g. Existence of
design reviews, test specifications, test reports, etc.

- **Consistency:** Ensures the dependency links between each stage in the overall
  information model are acknowledged. e.g. That code is not produced before being
designed. Verification & validation and risk assessment artifacts are dependent on
each other and on technical artifacts.

- **Activities:** Concerned with the agents involved and the methods used by those agents.
  Consideration is given for agents’ responsibilities, and in addition their skills, and
  experience, which are critical to the efficacy of any activity. For methods,
  consideration is given for information representation (descriptive), and the technique
  used to transform the information into the representation (prescriptive) e.g. formal
  transformations are more likely to be complete and consistent than informal ones,
  although the respective agents’ expertise is also very important.

The ASPEN tool, runs on UNIX platforms and on PCs under MS-Windows. Figure 6.11 illustrates the
ASPEN tool architecture.

![User Interface Diagram](image)

**Figure 6.11:** The ASPEN Tool Architecture

It provides various editors for creating and tailoring the *Methods, Standards, and Applications Knowledge Bases*, and the *Information Framework* of a process model. The process model can be visualised graphically in various forms, for example, as Role Activity Diagrams. Since the paper provided no screen shots of the ASPEN tool in action, an examination of the user interface itself could not be conducted.
6.5.4 Applying the Maturity Model to CASE?

As mentioned previously in section 2.4, new tools and methods introduced to an organisation with process maturity at only the Repeatable stage may have damaging effects on the performed process. However, this observation may appear superficial, since CASE tools vary greatly in their applicability and their impact on a development process. A wider definition of the scope by which CASE tools can impact a development process is required. There are many CASE tools which focus on simple, mechanistic tasks, impacting only at the lowest levels of a process, affecting perhaps the work process of a single user.

Extending this argument, this author believes that the Capability Maturity Model could be theoretically applied to a CASE tool: for example, an IPSE such as CLIME [Sivess 1996a] could be classed at the Managed level, since it deals with complete lifecycle management. In general therefore, any CASE software (especially software-oriented) could, or perhaps should, be designed with this framework in mind, clarifying at which organisational maturity it is targeted. After all, an organisation at the Repeatable level would have little use for a fully integrated project support environment, since it would not be at a stage where it could integrate it successfully nor possibly even understand it. This argument then presents the question: how much managerial power should a software process-oriented CASE tool be awarded? Indeed, is it right for such a tool at the Managed level to have full control over the software management process?

6.6 Summary

It has been emphasised that a high focus of requirements throughout the software process can lead to a substantial decrease in project risk. By increasing the attention the requirements analysis phase receives, a greater level of project understanding is attained which consequently reduces uncertainty concerning a project. Partly, this can be achieved by realising the conceptual difference between non-functional and functional requirements. Also, by ensuring that particular attention is given to NFRs, a greater and more predictable level of quality can result, and these NFRs can thereby form an effective quality control check during the project validation phase.

As two examples, the DAIDA and ASPEN frameworks have been examined to show how they cater specifically for requirements handling. Chapter 8 will introduce a framework which attempts to consolidate elements from the fields of process modelling and requirements analysis, and provide a basis for handling requirements from an organisational perspective. The method of representing requirements
hierarchically given in section 6.3.1 forms the foundation for this framework, which demonstrates how a contractor/sub-contractor model of project construction can handle requirements hierarchically.

7 Validating Romula with a Pseudo-Realistic Process Model

This chapter introduces the first large example process which was used to validate Romula. Section 7.1 will discuss the aims and issues of validating Romula, and section 7.2 will discuss the methods and results of this validation. Section 7.3 will validate the process model used to validate Romula, and discuss the methods and results of this validation.

7.1 Introduction to the Validation Process

The aim of this testing phase was twofold. Firstly, to demonstrate that the Romula animator could successfully, within the context of a small-scale engineering environment, enact a process model which paralleled the software development process of a small project. Secondly, to validate the pseudo-realistic process model ProcMod as a viable software development process model. ProcMod assumed the role of a 'hard' management policy; the process being enacted was strictly followed and seldom deviated from. This disciplined approach ensured that both the animator and ProcMod were harshly evaluated for their applicability.

Although ProcMod was never designed to be an all-encompassing and realistic process model, it was still evaluated for its suitability in software engineering. Of course, such an aim would be over-ambitious in any case, since any concrete-level process model with project-wide scope would not be applicable for all software projects; the domain of software engineering covers a very wide variety of software practices for a wide variety of software. For example, a process model developed for the engineering of database systems would require more bias towards the understanding of data, whilst military projects would require great emphasis on utilising rigorous requirement and design methodologies, like SSADM [Downs et al. 1988].

Whilst validation was itself an interesting activity which greatly benefited Romula and ProcMod, during this phase a much greater understanding of the software development process yielded some interesting insights into the modelling issue of process deviance, which has been previously discussed in section 3.4.
7.2 Validating Romula using 'ProcMod'

This section will introduce ProcMod as a pseudo-realistic process model which was used to validate the Romula process modelling language and animator, and explain the validation method which was employed.

7.2.1 Introduction to ProcMod

In order to test the Romula animation tool appropriately, a well-defined, complex process model was developed which incorporated all required aspects of process models. Process Model version 5.2, or ProcMod, was the result. This is given in appendix B.

The aim was to produce a requirements-oriented process model, focusing on how functional requirements (FRs) and also non-functional requirements (NFRs) are created, fulfilled, and eventually validated. What resulted was a model which deliberately incorporated features of many differing models, frameworks, and methodologies.

An effort was made to incorporate as many complex features of process models as possible into the model. This was to ensure that ProcMod was capable of adequately testing Romula's capabilities. This meant that in terms of realism, ProcMod has its faults (see section 7.3.3). However, the aim was not to produce an overly realistic model which could be utilised for a specific organisation, but to produce a pseudo-realistic model with which to test Romula.

7.2.2 Romula Validation Issues

Conducting validation of Romula was very straightforward. If any problems were encountered during the enactment of ProcMod, they were to be documented. Such a problem would fall into one of the following categories:

- **Inability to model ProcMod in Romula:** The possibility exists that the Romula language may not be able to express parts of ProcMod properly, because it has not the functionality to do so.
• **Inability to provide a clear and helpful enaction of ProcMod:** The possibility existed that the Romula animator could not clearly and effectively express ProcMod, in that it could not appropriately detail the current state of the model, or suggest the next tasks that needed to be performed.

Of course, as with verification, to some degree Romula had already been validated. However, up until this point ProcMod was executed on the animator in parts as it was developed, and never in its entirety.

### 7.2.3 The Validation Method

Initially, ProcMod was translated into the Romula language so that it could be enacted in the Romula animator. It was observed how easily and efficiently ProcMod could be modelled in Romula, and how clearly and effectively it could be enacted. Since Romula had already been verified, the latter proved not so much an issue of reliability, but more an issue of aesthetics: how well does the user interface aid the modeller in enacting a process model? Hence, validation of Romula's suitability in enacting a large-scale process model consisted of answering the following questions:

• **How clearly is the model represented by Romula?**
  
  This essentially involved evaluating how effective the language was at representing the model, and how difficult the translation from RADs to Romula proved to be. Are RADs being properly represented by Romula? Are we having to sacrifice any detail when translating into Romula?

• **How effective was the animator in allowing the user to control the model?**
  
  Does the user interface at any stage during enaction impede the modeller from performing a desired action? Is it unclear at any point how to operate the user interface? Is the state of the model at any point during enaction unclear? Is the desired information being displayed?

Therefore, evaluation of the language was concerned with *clarity* and *completeness*, and evaluation of the user interface was concerned with *clarity* and *effectiveness*.

### 7.2.4 Results of Validating Romula

This section details the results obtained by validating the Romula language and animator.
7.2.4.1 Romula’s Ability to Represent ProcMod

Since Romula was essentially based on the Role Activity Diagram and its modelling philosophy, translating ProcMod into Romula proved to be a straightforward process. For an example, see Figure 7.1.

![Figure 7.1: Fragment of ProcMod, involving Requirements Analyst and Customer roles](image)

This illustrates a Requirements Analyst role performing initial requirements analysis tasks, with the Customer role assisting with ’Management / End User Interviews’. This example demonstrates two of the more sophisticated features of RADs in action: concurrency and interaction.

The corresponding Romula segment for the Customer role is shown in Figure 7.2.

![Figure 7.2: Corresponding Romula Customer role segment from Figure 7.1. depicting how this role handles an interaction from another role](image)

These example demonstrates the ease with which RADs can be translated into Romula. Usage of extraneous programming language syntax common in other modelling languages is here less liberally used. Modelling languages based on programming languages are fundamentally tied to the syntax of that language. For example CSPL [Chen 1997, Chen and Tu 1994a, Chen and Tu 1994b], being based on Ada...
and the unix shell, inevitably inherits syntax from both of these languages: it is a *programming*-based modelling approach. However, Romula is far more modelling-based; the language is far less compromised by concepts from the programming domain. An example of this is task concurrency: tasks are referenced simply and directly by name. Another example of this is interaction. As discussed previously in section 4.9, the specification of interaction is quite clear and intuitive, and clearly related to modelling, whereas if we had directly adopted appropriate programming concepts to achieve this, we may have borrowed ideas from MPI, or the TCP/IP socket approach. This would probably have led to a more complex and unintuitive interaction mechanism.

These examples also demonstrate the RAD-capability-to-Romula-capability correspondence discussed in section 4.3. As much as possible, for every single feature in RADs, there is a corresponding single command in Romula. This is demonstrated by interactions, 'DO' tasks, conditions, etc. This modelling philosophy permits models to be constructed in Romula that are as complete and almost as easily understood as their RAD counterparts.

```plaintext
# ...
CONCURRENT

TASK 'Modelling' :
  DO 'Select appropriate modelling approaches'
  DO 'Construct system models'
  ENDTASK

TASK 'Interviews' :
  INTERACT WITH 'Customer' FOR 'Interviews'
  ENDTASK

TASK 'DocumentAnalysis' :
  DO 'Document/Data Analysis'
  ENDTASK

TASK 'ProblemPartitioning' :
  DO 'Problem Partitioning'
  ENDTASK

ENDCONC

DO 'Represent system model in Telos'
# ...
```

**Figure 7.3:** Corresponding Romula *Requirements Analyst* segment from Figure 7.1, depicting usage of concurrency and interaction

Although Romula can reference other roles by *instance* as has been demonstrated in section 4.4.6, via a role list variable (e.g. `$customer`), in these examples roles need only be referenced by *type*. In this case, the modeller is free to use a level of abstraction that is required. The syntax and general flexibility of Romula allows the modeller to represent a model, its activities, and its interactions in as much syntactical detail as is necessary, with a minimum of linguistic 'garnish'.

127
It is for these reasons that the translation of ProcMod into Romula proved a simple and straightforward process. The Romula representation of the version of ProcMod used for validation can be seen in appendix B.

7.2.4.2 Romula’s Ability to Enact ProcMod

This validation activity proved straightforward. Since the progress and state of each role (and role thread) is clearly represented by a separate window, it is clear most of the time what the state of the entire model is. However, it was also concluded that the effectiveness of the user interface was dependent on the size of the computer monitor on which it was displayed. Smaller monitors which could handle only the medium resolution of 1024x768 pixels often got cluttered with role thread windows, and it was often unclear what the state of the ProcMod model was, due to its size and complexity. It was concluded that a resolution of 1600x1200 on a monitor of 21” in size is ideal for displaying a model of such scope as ProcMod, but the user interface should also be revised to reduce the size of the role instance windows.

A more fundamental concern was that at its current state of development the Romula animator has no state storage or retrieval facility for enacting models. Although this was not one of the requirements of the animator prototype, since the main purpose of the prototype was to demonstrate proof of concept, the fact that the enaction of ProcMod had to be completed in one session proved a distinct inconvenience.

7.3 ProcMod: A Valid Process Model?

We have demonstrated in section 7.2 that Romula is capable of modelling and executing an example process model which attempts to encompass a satisfactory level of realism and scope. However, it may be the case that this level of realism and scope is not sufficient for the task of validation. To help alleviate this concern, and thus ensure a quality validation process, we may take validation further. By attempting to validate and hence award credence to the validation process model itself, by establishing the realism of this model, we are further justifying the claim that Romula can successfully model real processes.

7.3.1 ProcMod Validation Issues

To ensure ProcMod was stringently validated, a well-defined validation method was created, which is detailed in the next section. However, the validation process was conducted by a single individual assuming the responsibilities of all roles within ProcMod, and not within a more real, larger-scale
engineering context for which ProcMod was designed. Thus, in order to appreciate the value and \textit{validity} of the information gained from the validation process, it is important to understand the shortfalls of such an individual-based approach, and how such an approach may affect the results of the validation process:

- Since all roles would be 'acted' by a single person, that person would obviously understand all the problems and issues that arise during the project. That individual would intrinsically be aware of the ramifications of a problem for all project roles. This overall project awareness, (inherent in the individual-based validation process), would be impossible within a real (larger scale) software engineering environment, upon which ProcMod was modelled.

- The individual-based validation approach assumes a single entity communications model, in that there are simply no paths of communication. Multiple \textit{roles} exist, but the tasks of each are conducted by one person, so the inter-role communication model is non-existent. Of course, with large-scale software engineering environments, the inter-role communication model would be quite complex. For example, clarification regarding requirements may be required for a \textit{Designer} role from a \textit{Requirements Analyst} role. This is obviously a less than trivial exercise for an individual enacting all project roles. However, within a larger scale software engineering environment, different problems will inevitably arise with different individuals who have different responsibilities (within their role group). Such problems would require interactions between roles that would be necessary, and therefore would not occur, in individual-based validation.

These predetermined disadvantages necessitated the adoption of an appropriate 'attitude' to ensure validation was conducted as fairly, and as realistically, as possible. Therefore, a measure of respect was required for roles as isolated project groups. Each role in ProcMod was acted as a completely separate entity, with its own agenda, and project responsibilities. In addition, the knowledge acquired by each role concerning the project was documented separately from other roles. Thus, as each role was acted in turn, a separate project knowledge base emerged for each role, containing information on the following:

- **Role's State:** Process information was accumulated on what had been done, and what task was currently being performed.

- **Role's Project Knowledge:** As project tasks were completed by each role, details of project knowledge (e.g. documents, models, etc.) were recorded.
Therefore, a multi-person validation process was defined and simulated as effectively as time constraints allowed.

### 7.3.2 The Validation Method

A simple software development scenario was created to validate ProcMod by enaction. The scenario has the following two main objectives:

1. Ensure that all process paths within ProcMod were followed and enacted.
2. Ensure the model was 'balanced'. During enaction of the model, situations may arise when a role cannot continue and is forced to halt until either other task(s) have been completed concurrently by itself, or other role(s) are also ready to enter a specified joint interaction.

Creating a complex scenario would not have been within the scope of this thesis, and probably would not have yielded proportionately more insight into ProcMod's validity. A more complex software scenario would have detracted from assessing the model, and would have shifted the focus onto the product being created, which was not the aim of validation.

The scenario was to design and create a simple desktop calculator for a small company that already uses computers for word processing under Windows 95. The initial Project Plan was to develop a calculator application that:

- Possesses simple arithmetic functionality (+, -, *, /).
- Runs under Windows 95 and utilises a simple graphical user interface that appears and functions similarly to that of a real calculator.
- Is frequently used.

Although oversimplified, this Project Plan is simple and concise, and was thought to provide adequate scope for software development. Without wishing to pre-engineer the product before the project was begun, it was anticipated that this scenario would sufficiently test ProcMod. Solid requirements and design would be necessary for both the arithmetic functionality and graphical user interface, perhaps represented as individual modules, and for coupling between each module.
Separately for each role, the following steps were taken when acting the process model. At each point, (or as a new state in the process model was reached):

- State task performed.

  OR

- State reason for not performing action.

  OR

- State reason for 'backtracking' to a previous point in the process model.

- State reasons for inappropriateness of model, if required. Such a reason could be 'task stagnation', where a task cannot be started until the other roles which are also part of this interaction are ready to enter the interaction.

Additionally, following each interaction and condition:

- State consequences for each role.

- Note if 'interactive stagnation' has occurred. This type of 'Model stagnation' can occur when the role cannot continue until the interaction has been completed, and has had to wait for an inordinate amount of time before it is able to enter a post-interaction state.

- Note if 'conditional stagnation' has occurred. Stagnation can also occur when a condition state is entered and a decision cannot be made until other activities elsewhere in the model have been completed.

These types of stagnation can cause process bottlenecks when enacting the model, perhaps preventing multiple tasks from being started which could well have been started earlier. In many cases stagnation can occur naturally and not inconveniently halt the model, for example, the Project Manager role may be required to wait significantly for a progress report to be completed by another role and received. This clearly would not adversely affect the enacting of other roles, since the Project Manager is the role that is dependent on the other roles for progress reports. However, the possibility existed that stagnation could occur within one role. For an example of such a badly structured model, see Figure 7.4.
In this example, three tasks are executed concurrently; A, B, and C. Let us assume that task A requires much more time to complete than the other two. Additionally, let us assume that this model is part of a role which, when instantiated, is enacted by several individuals, and that task D only requires tasks B and C to be completed before it can begin. In this instance, when tasks B and C have been completed, D cannot be started until A has also been completed. A waste of the effort resource thus occurs: those responsible for tasks B and C are idle, whereas they could begin task D. A version of this model which would suit this scenario better is shown in Figure 7.5.

Here we can observe that the model is now more balanced; D can now be started whilst A is still being completed.

### 7.3.3 Results of Validating ProcMod

Following validation, it was concluded in general that ProcMod provided good support for guiding the development of the calculator scenario, and it never became necessary to deviate from the model. Following the development method proposed by ProcMod resulted in a functional piece of software which met its requirements.

However, it exhibited some minor unrealistic behaviour which detracted from its enactability, which is discussed in section 7.3.3.1. During the validation process, certain possible methods to improve ProcMod
became apparent each of which would require fundamental redevelopment of the model. These are addressed in section 7.3.3.2.

7.3.3.1 Observations Concerning ProcMod

All observations given should be prefixed with 'Realistically, . . . , since realism is the testing subject of the process model.

1. A Project Manager would be part of the internal technical review process during the design phase.

2. 'Selecting Appropriate Modelling Approaches' (requirements phase) may include Telos implicitly.

3. 'Management/End User Interviews' (requirements phase) should precede 'Construct System Models': cannot construct models without information concerning the application domain.

4. 'Problem Partitioning' (requirements phase) should be performed after 'Mngmnt/Interviews' and 'Data/Doc. Analysis'. A more structured ordering of this concurrent section is required.

5. Functional Behaviour should be included concurrently in the later stages of Requirements; at the moment it's more or less implied by the inclusion of Telos.

6. Some projects may not use some software development sub-processes, for example, Doc/Data Analysis, as in the calculator exercise. Need to be able to skip processes.

7. Goal-Satisficing Methodologies need to be performed for practically every design decision to justify them with respect to requirements. i.e. Design Decision(s) made then Select appropriate GSM then Ensure NFRs remain Satisficed. Perhaps Design could be restructured to model this sequence.

8. Loopbacks need to be refined. Requirements and Design satisfaction queries imply all previous sub-processes need to be redone in the event of an unsatisfactory Requirements Specification or Design Specification. Of course, this would be unlikely.

9. Better design process specification; modules may require a more formal definition than pseudocode. The Design to Component Engineering crossover in the calculator exercise highlighted the need for better function parameter definition, necessary for uncomplicated integration.
10. Coders may need to communicate during implementation. This may be to ensure better module coupling, or for more managerial concerns such as walkthrus.

11. Integration Engineer should be started as late as possible. The current version of the process model starts the Integration Engineer in Requirements, so they can collect the Validation Test Criteria. However, a more efficient use of human resource would be to start this role at Integration, then pass relevant documentation.

12. Perhaps the Integration Engineer (obviously having experience with integration language(s)) should fix minor problems (only minor integration problems, not fundamental flaws!) themselves.

13. More than one Component Engineer could be started simultaneously for debugging purposes in Integration, especially within large-scale projects. Currently, the Integration Engineer starts a single Component Engineer to debug all problems found at Integration. Workload should be better divided and delegated.

Incorporating most of these changes into the model resulted in version 6 of ProcMod (see next section). However, implementing some of these amendments (notably 6, 7 and 8), would have required a fundamental redesign of the model, essentially altering ProcMod’s view of the development process itself. For example, amendment 7 would significantly alter the design process in the Designer role, shifting the design approach to a design decision-oriented process. i.e. Design decision made, then traced to requirements to ensure they remain valid, and if so, incorporate that decision into the product design. Currently, ProcMod is entirely task-oriented. For example, in design, design decisions are made implicitly within the preliminary design phase ('Translate Requirements and TDL into Data Design' and 'Translate Requirements and TDL into Software Architecture') and within the detailed design phase ('Refine S/W Arch. & Data Design into lower-level modular segments' and 'Interface Design'). Such a fundamental paradigm shift in design would leave the other development phases of ProcMod inconsistent with this approach; requirements analysis performed within the Requirements Analyst role would also have to become decision-oriented, and additionally the interactions between roles would have to be updated. Of course, completely redesigning ProcMod in this manner would render the other amendments above irrelevant, and ProcMod would have to be revalidated.

Amendments 6 and 8 also could not feasibly be incorporated into the model. Amendment 6 would require a condition for every task performed. Including 'Do you wish to perform <some task>? Yes or No' for every single task would make ProcMod profoundly cluttered and ultimately confusing. For this reason the issue of intentionally deviating from the set model was investigated separately (see section 3.4). Similarly, incorporating amendment 8 would require separate conditions for every previous task within a phase. e.g. 'Is Design satisfactory? Yes or No'. If it was not, the following condition would be entered: 'At which
Another concern with ProcMod was the manner in which process re-engineering was handled. See Figures 7.6 and 7.7. Figure 7.6 illustrates how re-engineering design (if required) is modelled in ProcMod for the *Project Manager* role. However, this approach is distinctly untidy, and needs to be duplicated for every other role to ensure that their states are also accordingly altered if re-engineering design is deemed necessary. In addition, this untidy process also needs to be duplicated for every separate development phase. A better RAD approach which covers such an eventuality is not detailed in Ould [Ould 1995], nor could one be devised. A possible approach which may achieve this more satisfactorily could be a method which somehow allows a single role to alter the state of another remotely. This type of inter-role 'interference' happens frequently in the real-world: time constraints, for example, sometimes necessitate a superior to conduct a premature (yet normally ill advised!) shift from the requirements phase to the design phase within a software project.

![Diagram of Project Manager role](image)

**Figure 7.6:** Portion of the *Project Manager* role, depicting a possible re-engineering of the design phase

**Figure 7.7:** Portion of the *Project Manager* role in Romula, depicting how design re-engineering is represented

In any event, this unfortunately meant that the corresponding Romula code segments dealing with re-engineering phases inherited this clumsy approach. As an example, the Romula translation of Figure 7.6 is given in Figure 7.7. Since the Romula representation of ProcMod was to be strictly based on the RAD version of ProcMod, any better methods of performing this activity in Romula could not be utilised. Therefore, whilst Romula inherits the intrinsic advantages and strengths of Role Activity Diagrams, it can also inherit its disadvantages. Although, fortunately, some of these major shortfalls have been recognised and addressed (for example, the handling of role instances, discussed in section 4.4.6).
7.3.3.2 Possible Methods of Process Improvement for ProcMod

Following validation, it was observed that four possible paths of process improvement exist for ProcMod:

1. Construct improved process model applying same level of detail with noted improvements.

2. Construct improved process model at a higher abstraction level; more generalised, less restrictive. In addition, this implies more generalised terminology.

3. A hybrid of methods 1 & 2. Usage of a higher-level, more generalised structure, but with conditional detail; detail which could be enacted based on the result of a condition to do so.

4. Build a process model (possibly incorporating aspects from methods 1, 2 & 3) based on the data dependency demonstrated in the process model DFDs (see section 4.8.3).

It was decided to conduct process improvement method 1, which created version 6 of ProcMod. Although validation was performed by a single individual, ProcMod was not found to be too restrictive, and generalisation of the model (method 2) was deemed unnecessary. Appropriate use of concurrency allowed tasks to be performed as and when desired, and backtracking to a previous task was only performed once (during design). In addition, generalising ProcMod may have significantly reduced its usefulness in aiding the software development process. Therefore, since generalisation was thought unnecessary, the hybrid improvement approach suggested by method 3 was also discounted.

Method 4 suggested an entirely new and interesting way of structuring ProcMod, and was investigated separately. See section 4.8.3.

7.4 Summary

This chapter has firstly introduced the pseudo-realistic process model ProcMod which was used to validate the Romula language and animator. Developed in parallel with Romula, the model matured from a simple test model to a large-scale model exhibiting an adequate level of realism. It proved a valuable aid in ensuring the development of Romula remained focused on producing a language which was sufficiently real to be a viable modelling approach. By using an example project scenario to validate ProcMod, and hence validate the method we employed to validate Romula, we are further lending credence to the argument that Romula is a valid process modelling approach.
As a result of validating Romula, we have:

- Demonstrated that the translation from a substantial, complex Role Activity Diagram to Romula is essentially a straightforward process.
- Demonstrated that there does exist a one-to-one correspondence between the features of RADs and the constructs of Romula, further illustrating the straightforward nature of the translation process.
- Further demonstrated that Romula is a modelling-oriented process modelling language.
- Stated that although enaction is a clear and straightforward process, there exists no method to store or retrieve the state of an enacted process model, and a substantial display monitor is required to display large-scale models clearly.

As a result of validating ProcMod, we have:

- Demonstrated that despite the fact that the method used to validate ProcMod was individual-based, many deficiencies were uncovered which led to the construction of an improved process model.
- Demonstrated that ProcMod was a good model for validating Romula.
- Illustrated by example that ProcMod is task-oriented as opposed to decision-oriented.

It is the first point that proves most important: we have effectively demonstrated that Romula can be used not only to define a process, but to aid in improving that process. For a software organisation, it is the process improvement which can occur at CMM levels 4 and 5, (see section 2.4), which is undoubtedly the ultimate tangible goal of defining a process model, which occurs at CMM level 3.
8 Validating Romula by Implementing a Requirements-Oriented Process Modelling Framework

8.1 Motivations

This chapter will introduce a framework which represents the second example of Romula may be applied to modelling processes. It differs from ProcMod by attempting to model processes and requirements across many organisational entities, as opposed to within the context of a single organisational entity. It achieves this by allowing each set of project requirements within a requirements' hierarchy (see section 6.3) to be associated with a process used to fulfil that set of requirements, and representing each requirements/process pair within a single definition. Each of the definition within the requirements/process hierarchy can then be associated with an organisational entity responsible for fulfilling those requirements. As will be demonstrated, representing this framework can prove a complex task. Section 8.1 discusses the motivations and origins behind this framework, whilst section 8.2 introduces the framework. Section 8.3 illustrates how the framework operates by simple example, and section 8.4 introduces and discusses two example frameworks and defines their processes using Romula.

8.1 Motivations

Software Engineering, at the most basic level, consists of translating a set of requirements into a product. But requirements often remain poorly represented, and consequently are only partially fulfilled, by undefined, ad hoc development processes. This chapter consolidates research from Requirements Engineering and Software Process Modelling and proposes a framework called AMMROF (A Micro-Modelled Requirements-Oriented development Framework) which allows requirements and associated development processes to be represented in a single definition. This framework allows an organisational infrastructure to be created and dynamically modified to represent requirement-based ties between component engineers. Two example process models will be presented which illustrate and briefly discuss issues with modelling within such a framework. Each of these models will be encoded in Romula to illustrate how Romula can cope with modelling such a framework, and demonstrate how this can prove to be a complex modelling task.

AMMROF is a method for representing project requirements with associated process models, which model the eventual fulfilment of those requirements. These models simulate the process of software development
from a perspective of completing requirements hierarchically, ‘micro-modelling’ the process of completing each set of requirements at each level in the hierarchy.

As previously discussed in section 6.2, the Requirements Engineering Process is often ill-defined and poorly supported. The DAIDA model and the ASPEN framework, as discussed in sections 6.5.2 and 6.5.3 respectively, help to promote the awareness and handling of requirements via well-defined process worldviews. The DAIDA CoNeX prototype software allows a project to be represented initially only as a requirements model, ensuring design issues are not imposed prematurely upon project requirements. The AMMROF framework follows a similar approach, only allowing subsequent phases in the software process (such as design) to translate and implement project requirements once those requirements have been fully specified and refined. By following such a strict protocol, superceding design, for example, can only be conceptually represented in these models in strict relation to the requirements model, ensuring design rationales can be suitably traced back to their corresponding requirements. Although the models differ in representation, it is this assumptive property that maintains the focus on requirements until design is complete.

Rayson et al [Rayson et al. 1999] describe the problems associated with recovering requirements information from legacy systems, and propose methods to reverse-engineer the initial business processes and conceptual models from legacy data. The framework proposed in this paper conveniently describes a method of detailing the requirements and their associated business process, preserving both types of information in a single framework whilst maintaining a core focus on requirements throughout the development process. In essence, a requirement is created to be fulfilled, and inevitably the fulfilment of this requirement is achieved by an activity. So why not define and associate both as a declarative entity?

8.2 The AMMROF Framework

AMMROF is designed to capture two distinct aspects of the development process:

- **Representation Framework for Requirements:** A concrete representation of the requirements in a hierarchical format. Each node of the hierarchy is essentially a requirements list, with an arbitrary number of sub-requirement lists. Note that requirements are not mandatorily software-related, they could be document-related, for example a certain type of user manual is a requirement of any software project.

- **Method used to Fulfil Requirements:** The subsequent development processes associated with the fulfilment of the defined requirements. Differing types of requirement, such as software, document, etc. may have differing methods of meeting
that requirement. Therefore, each given requirement list has an associated defined completion method. Each method may employ the use of various specialist roles involved in the completion of a requirement's process; designers, component engineers, librarians, etc. These methods can be captured as process programs in Romula, introduced in chapter 4. Once captured as programs, these process methods can be enacted to test their validity, or executed in a real software development environment as a process support tool to provide visualisation of the process being used.

Each separate 'node' in the hierarchy, therefore, contains two sets of information; a list of Requirements and their associated Process with which to complete those requirements. For clarity, these lists are referred to as Requirement lists, and lists refined from a given requirements list are referred to as Sub-Requirement lists. For more information on representing requirements hierarchically in lists, refer to section 6.3. Since we are modelling software project requirements and process, each node will be referred to as a 'project node'.

The sub-requirement lists of a given requirement list must first be completed using their associated predefined process before that requirement list can itself be completed. The Requirements segment details the specification of the requirement list, the Process segment holds the process definitions which deal with elaborating upon and completing that requirement.

The Requirements segment is defined as follows:

- **Requirement Definition List:** An abstract, initial list of requirements. This is the structure's 'motivator'; all process activities defined to complete a requirement must ultimately fulfil this statement.

- **Requirement List Description:** An elaboration of the Requirement Definition List, containing more detailed information concerning the demands of the requirements and perhaps including a requirements model to further aid understanding. Depending on whether it is necessary, the requirement list may be refined into sub-requirement lists, each constituting a lower-level node in the hierarchy. At a low abstraction level in a project hierarchy, like the engineering of a component, it may include elaboration of the requirements for a programming interface. At higher abstraction levels, it is more likely to include elaboration of functional requirements. All defined process methods are created to fulfil these requirements.
The Process segment consists of an ordered template onto which the process methods for fulfilling a requirement are defined. Once fully specified in a process programming language, the Process segment can be executed, allowing a hierarchy of project nodes to be created:

- **Requirement Analysis Method:** This process produces the Requirements List Description from the Requirement Definition List. The emphasis on this activity varies greatly with the type of requirement. In some cases, it involves an in-depth analysis of the requirement list, gaining an understanding of their depth and complexity. This would be required when engineering a component, and associated sub-components. Other cases where the importance of this stage is not a priority could be the creation of a document. However, in any case it is necessary to understand at least satisfactorily what is required before the process of fulfilling a requirement is attempted. It is at this stage that it is decided whether or not refinement of the requirement list into sub-requirement lists is necessary.

- **Sub-Requirement List Instantiation:** This is a generic stage which facilitates refinement of a requirement list into sub-requirement lists, if necessary. This part of the definition instantiates new nodes in the project hierarchy, hierarchically linked to the current node, based on the refinement model given in the Requirement List Description, if any. The creation of a sub-requirement list project node is therefore termed Sub-Requirement List Instantiation. A requirement list cannot be fulfilled until all sub-requirement lists themselves have been fulfilled.

- **Process Method:** This is the implementational core of the overall process definition. This stage describes how the requirements themselves are to be fulfilled by creating something that fulfils them. For example, in Romula, varying types of project engineer are utilised in a socio-linguistic manner to complete the set of requirements. Activities within this widely scoped category can include design, detailed design, and implementation. So, instead of modelling the entire process which creates and ultimately fulfils a plethora of requirements, the framework models the requirements and their associated process. Hence, this is effectively ‘micro-modelling’ the development process within the context of each set of requirements.

- **Sub-Requirement List Satisfaction:** Another generic process stage which effectively ensures all sub-requirement lists within lower-level project nodes have been completed (i.e. they have completed the Satisfaction Method stage) before proceeding to the latter phases of completion.

- **Integration Method:** When all sub-requirement lists have been fulfilled, some type of component integration may be required to complete that list of requirements.
Examples are the construction of a set of code modules which require integration, or a
document which is written in parts and requires compilation into a completed
document.

♦ Validation Method: Validation can play an important role after integration; once a
project node's component(s) are integrated with its sub-requirement node's
components, usually a validation process is undertaken to ensure the completed
component(s) properly fulfil their respective requirement(s) in the requirement
definition list. Examples are the validation of an integrated software system with
respect to its requirements, or far less stringent, checking the consistency of an
integrated document which was created in parts.

♦ Satisfaction Method: This generic activity is simply required to notify a project
node's parent project node that it's list is completed and fulfilled. Viewed recursively,
this method notifies the parent Sub-Requirement List Satisfaction method of its
completion.

Note that these methods are split into two types: Generic and Engineering methods. The generic methods,
Sub-Requirement List Instantiation, Sub-Requirement List Satisfaction, and Satisfaction Method, are
mandatory, and are identical throughout all types of project node. Without these methods, the project
hierarchy could not be created, nor could it be completed when the entire model is executed (or enacted) as
a process program. Much of the reason for prescribing the above structured definition format is to ensure
that these methods are not missed out or included ad hoc.

A requirement's Process Method can only begin once requirements are refined. This ensures that a
requirement list is at least better understood before fulfilment of the requirement list is attempted. Also
important is the Validation Method, the corresponding activity to Requirement List Analysis. This activity
ensures that whatever is produced by the Process Method (and sub-requirements) in response to a
requirements list is valid with respect to those requirements. These two software engineering activities are
common to all definitions, stressing the importance of creating artifacts which are consistent with their
requirements.

The engineering type of methods, the Requirement List Analysis, Process Method, Integration Method and
Validation Method represent the activities that should be carried out during software development. It is
within these methods that the actual process is captured. It should also be noted that differing types of
project node (e.g. software documentation, Component engineering) may place varying emphasis on each
of these methods. For example, software documentation may require only simple requirement analysis,
such as the creation of a contents page. Component engineering, on the other hand, may require in-depth
requirements analysis, leading to functional requirements decomposition, and identification and elaboration
of non-functional requirements. Likewise, validation for a software documentation project node may simply involve checking the adherence to the required contents, but also may involve some in-depth validation of the documentation with respect to the usage of the software.

It initially appears that a corresponding design method has been left out. Although implicit, design is included within the *Process Method*. There are reasons for this. Firstly, we do not wish to promote the misconception that this framework in any way represents a real process model. The process segment of the project node is simply a *template*, onto which a real process model may be detailed. Secondly, this framework is *requirements-oriented*; we wish to emphasise that requirements are the driving force behind software development - the prime motivator. However, we do not wish to lessen the importance of good design, which nowadays is simply a necessity when developing software. Lastly, we are not concerned with design *per se*, since the framework operates at a requirements level of abstraction. *We are* concerned with requirements, how they facilitate the construction of the project node hierarchy, and how they eventually are fulfilled by the emergent product.

### 8.3 An Example of the Framework

This flexible framework permits arbitrary activities of varying complexity to be described, although a good understanding of their practice is clearly required before such a process definition can be made [Hahn *et al.* 1990]. This is achievable by the fact that the *Process Method* may involve many abstract software engineering activities, perhaps detailed design, component engineering, etc.

Figure 8.1 shows a hypothetical 'snapshot' of a simple example instantiation of the project hierarchy for a simple arithmetic calculator. In this example five project nodes, with their respective requirement lists (*R1* to *R5*), have been created hierarchically. In reality, of course, the requirements definition list and refinement would produce a much larger set of requirements, and hence a larger hierarchical representation. Also, analysis of the requirement lists would be far more extensive. However, the contrived simplicity provides an easier frame of reference.
Each of the requirement lists are defined as follows:

- **R1**: Develop a simple arithmetic desktop calculator
  
  Produce appropriate documentation for use.

*Elaboration & Refinement:*

- **R2**: Document calculator functionality
  
  Document calculator operation

- **R3**: Produce calculator software with simple arithmetic functionality

- **R2**: Document calculator functionality

  Document calculator operation

*Elaboration:*

  Describe calculator operation in terms of functions

- **R3**: Produce calculator software with simple arithmetic functionality

*Elaboration & Refinement:*

- **R4**: Design user interface
  
  Implement user interface

- **R5**: Design calculation engine with simple arithmetic functionality
  
  Implement calculation engine

- **R4**: Design user interface to provide access to the functionality of the calculator

  Implement user interface.
Elaboration:
Create user interface models with respect to calculator functionality and usage
Design user interface with respect to models
Decide on implementation method of user interface
Implement user interface

♦ R5: Design calculation engine with simple arithmetic functionality
   Implement a calculation engine

Elaboration:
Design calculation engine which exhibits simple arithmetic functionality
Implement calculation engine

The current state of the requirements hierarchy is as follows:

♦ R1 is the top-level requirement. This is an initial project requirement, so its completion would signify the completion of the project. It is waiting for the documentation and software to be completed before they can be integrated into a final product.

♦ R2 has started a Technical Writer, (TEC), to complete the user documentation. It must be noted that there may be other roles involved at later stages of document development before the requirement is fully satisfied.

♦ R3 is waiting for the user interface and the arithmetic engine to be implemented before they can be integrated into a final piece of software.

♦ R4 has finished implementation and therefore the user interface module has been completed. It has fulfilled its requirement definition, and requires integration with its parent requirement's component.

♦ R5 is still involved in an implementational activity, and has started two Component Engineers, (CE), to program modules which will form the arithmetic engine which will be integrated with the user interface by R3.

This example demonstrates how different types of requirements may be represented in such a framework. It should also be obvious that the example has illustrated a significant advantage of this approach: the state of the entire project is clearly visible. We can quickly ascertain that the user interface has been completed, and the documentation and software still remain to be completed. In addition, by examining the state of a
requirement’s process program, its state of completion can be easily determined; R5 for example is still in the implementation stage, while R3 is waiting to integrate the completed implementations of R4 and R5.

From a component construction perspective, the framework also provides modularity. Each project node can arbitrarily be engineered by separate real-world entities, depending on the scale of engineering required to complete the project:

- **Small-Scale Engineering:** If we assume that the desktop calculator example is to be designed by one single project team, project nodes may be handled by single individuals, or small groups of people. One of these mini-groups, specialising in software documentation, may become responsible for the project node concerned with documenting the software (R2). Other small groups may deal with the project nodes dealing with component engineering, etc.

- **Medium-Scale Engineering:** On a medium scale, large project teams or entire departments within an organisation may be required for a project. Similarly, a project team specialising in software documentation may be required for project node R2, and an entire department may be required for dealing with the component engineering project nodes R3, R4 and R5.

- **Large-Scale Engineering:** On an even larger scale, an organisation may wish to sub-contract some of the project hierarchy to other companies. Contractors specialising in implementing desktop user interfaces may be given the responsibility for project node R4.

Of course, a mix of these perspectives is certainly possible. R3 may be sub-contracted to another company, whilst R4 and R5 are dealt with internally within that company. Note that however the project nodes are dealt with, the entity that deals with the top-level project node is always the entity ultimately responsible for producing the final product.

### 8.4 Encoding Example Frameworks Into Romula

This section demonstrates how the process part of the AMMROF framework can be represented in Romula. A simple framework and a more sophisticated framework are given as examples of how this can be achieved.
8.4.1 A Simple Hierarchical Process Definition

We can demonstrate how a simple process model can be expressed in the framework by considering a simple contractor/sub-contractor example. Consider the RAD example of a simple contractor in Figure 8.2. This 'template' definition, at its current level of abstraction, does not detail the type of component engineering it performs. This allows us to demonstrate the 'mechanics' of the framework more easily.

The example follows the large-scale engineering perspective described in section 8.3, and depicts a contractor receiving a list of requirements to fulfil, perhaps to create a software component. Here we can see all the types of method in action. The contractor conducts Requirement List Analysis; analysing, elaborating, understanding, and refining that requirements list. Then, having further understood requirements and refined them into sub-requirements, the contractor creates an arbitrary number of sub-contractors, passing a list of sub-requirements to each (Sub-Requirement List Instantiation). Note that the contractor, by instantiating sub-requirement lists, has created sub-contractors to deal with these sub-requirements. Simultaneously, the contractor carries out his own Process Method, designing and implementing his own contribution to the component he is producing. When all the sub-contractor's components have been gathered (Sub-Requirement List Satisfaction), they are integrated (Integration Method) with the contractor's contribution to form the final component, which is validated (Validation Method) to ensure it meets requirements. Finally, the contractor notifies its parent contractor of its completion, returning its component.

![Figure 8.2: Simple contractor example as a Role Activity Diagram](image-url)
Note that this is a recursive definition; the creation of subcontractors from this model simply creates a new instance of this contractor model. In effect, an actual instantiation of this contractor/sub-contractor structure for a product would be hierarchical, and unsurprisingly, it would also be analogous in structure to the requirements hierarchy for that product. This contractor model could easily be applied as a process definition to the calculator framework example given in section 8.3.

8.4.2 Encoding the Simple Hierarchical Process Definition in Romula

Figure 8.3 is a Romula representation of the simple contractor model discussed in section 8.4.1.

This recursive definition of the Contractor role definition is quite straightforward. First, the contractor receives a list of requirements from its parent contractor, and conducts Requirement List Analysis on those requirements. Next, two tasks are started in parallel; one to deal with the contractor's Process Method, (a simple design and implement affair in this example, but could be more descriptive if required), and the second to allow the creation of sub-contractors (SubRequirementListInstantiation) and to deal with receiving their returned components (SubRequirementListSatisfaction).

There is an important difference between the contractor model RAD in section 8.4.1 and this Romula representation. The RAD representation specifies an arbitrary number of sub-contractors concurrently in its SubRequirementListInstantiation method in an arbitrary fashion. However, Romula cannot do this, since concurrent threads need to be explicitly named. Instead, Romula allows sub-contractors to be created by the user until no more need to be created. This is achieved by using Link and ExpectLink. Although we are using a variation on the Goto command by using Link, modelling real-world processes often requires this type of construct [Ould 1995]. In addition, Romula's Link and associated ExpectLink use reason-codes in a semantic manner which clarifies their use.

Romula interacts with its numerous sub-contractors using the interact interaction to specify a simultaneous group interaction with all roles specified, in this case, in a Romula instance list.
Figure 8.3: Romula representation of the simple contractor given in Figure 8.2

By executing this process program on the Romula interpreter we can demonstrate a simple contractor/sub-contractor model in action, creating sub-contractors and sub-sub-contractors, each performing various software engineering-related activities to produce their components. More importantly, we can observe the process of requirements flow: analysis, instantiation, satisfaction, and validation, although we cannot at this stage represent the requirement lists themselves.

However, this model has certain flaws which characterise it as unrealistic, which can be illustrated with certain simple scenarios:
1. A contractor decides to terminate its business with a sub-contractor prematurely.

2. A contractor, constrained by a deadline, or reacting to a sub-contractor's cost or time overrun, demands the sub-contractor 'comes up with the goods' almost immediately so it can complete its own list of requirements.

3. A sub-contractor decides to terminate its business with its parent contractor prematurely.

4. A sub-contractor, due to an unforeseen circumstance, wishes to terminate its business with its parent contractor, but supply the work it has completed to date.

5. The component fails validation and needs to be re-engineered.

6. A contractor belatedly decides another sub-contractor is necessary to perform a task after it has defined its sub-contractors and they have returned their components.

7. The sub-contractors return their components at differing times.

8. A contractor decides to change or update a sub-contractor's requirements.

Such feasible scenarios which are not catered for by the initial contractor model unearth the obvious flaws in its design, the first seven of which are addressed in the improved model in section 8.4.3.

8.4.3 A Realistic Hierarchical Process Definition

An improved contractor model is shown in Figure 8.4. Unfortunately, since RADs do not incorporate notation for describing interactions between instances of roles, additional notation is necessary to help define and clarify the model.

In explanation, the first two concurrent tasks (from the left in Figure 8.4) deal with a contractor's parent contractor, the third conducts the contractor's process method, and the fourth and subsequent concurrent tasks on the right deal with the contractor's sub-contractors. To aid clarity in understanding this self-referential, recursive definition, the interactions are numbered to illustrate how and where a contractor may interact with its sub-contractors or parent contractor to pass or receive a material (object created within the development process):

1. **Requirement List Passing and Receiving:** Where a contractor passes a sub-requirement list (refined from its own list of requirements) to a sub-contractor. Once business with the sub-contractor has begun, the sub-contractor initially receives this list of requirements, therefore performing the $SubRequirementListInstantiation$ method.
2. **Intermediate Component Passing:** When a contractor requests an intermediate version of a sub-contractor's component, the sub-contractor must be able to supply the latest version of its unfinished component almost immediately.

3. **Final Component Collection:** The contractor simply waits for a sub-contractor's component to be delivered. Note that this interaction is also used by a sub-contractor if that sub-contractor wishes to terminate its business with its parent contractor. It prematurely returns its unfinished component as if it were returning its final component, before killing all subcontractors and ending its relationship with its parent contractor.

The two main additions to the model, therefore, are sub-contractor management and parent contractor management. They enable a contractor to control its business relations with employees and clients. The four sub-contractor management activities exist to manage a contractor's sub-contractors individually:

- **Request Intermediate Component from Sub-Contractor:** The contractor can pre-emptively request a sub-contractor's component. This simply involves an interaction (interaction type 2 in Figure 8.4) which requests that component. At a lower hierarchy level, the sub-contractor accepts the request and sends the incomplete component. Observe that the sub-contractor must loop this request, since the parent contractor may make many such requests, or indeed none at all.

- **Request Intermediate Component from Sub-Contractor and Kill Sub-Contractor:** Identical to the activity above, except that the relation with the sub-contractor is killed.

- **Kill Sub-Contractor:** The contractor simply wishes to terminate relations with a sub-contractor by killing its instance in the enacting process model.
The contractor may make as many requests to as many sub-contractors as desired, and this management capability addresses problems 1 and 2 illustrated in the previous section. The flip-side to these activities are the parent contractor management activities:

- **Send Incomplete Component to Parent Contractor and Terminate Relations:**
  The contractor wishes to send pre-emptively its incomplete component to its parent contractor and terminate, so it produces a 'makeshift' component. An interaction (type 3 in Figure 8.4) is used to send this unfinished component to its parent contractor as a final component, as if the sub-contractor had actually completed its component and returned it naturally. All sub-contractor relations are killed, and the
contractor ends, although some components may have been returned from sub-contractors before this occurs, and therefore included in the 'makeshift' component.

- **Complete Component Naturally:** This simply allows the contractor to continue onto the Integration stage, not aborting the production of a component.

- **Terminate Business with Parent Contractor:** The contractor returns a 'null' component to its parent contractor, and just terminates relations with its sub-contractors and parent contractor.

These activities therefore address problems 3 and 4 in the previous section.

### 8.4.4 Encoding the Realistic Hierarchical Process Definition in Romula

Providing an explanation of the complete model in Romula would prove redundant, since the model contains much similarity. The complete model is given in appendix F, however, for completeness. Instead, the most interesting functions of the realistic model will be explained and illustrated. The overall structure of the improved Romula model is shown in Figure 8.5.

Problem 5 addressed in section 8.4.2 has been addressed; if the final component fails validation, the component is re-engineered via a Link to ExpectLink command pair.

Each of the ManageParentRelations, SubRequirementListInstantiation, SubRequirementListSatisfaction, and ManageSubContractors concurrent tasks will be detailed in the following sections. Note that all such concurrent tasks must 'complete' before integration can begin. Each task must incorporate a method to allow itself to complete (i.e. reach it’s EndTask), or the contractor would effectively encounter deadlock.
ROLE 'Contractor'
{
  INTERACT WITH '$CREATOR' FOR 'RequirementsList'

  EXPECTLINK FOR 'ReEngineerComponent'

  CONCURRENT
    TASK 'ManageParentRelations' :
      # manage business relations with parent contractor
      ...
      ENDTASK
    TASK 'SendIntermediateComponent' :
      # send an intermediate component
      # to parent contractor when requested
      ...
      ENDTASK
    TASK 'ProcessMethod' :
      # do simple process method for contractor's component
      DO 'Design'
      DO 'Implementation'
      ENDTASK
    TASK 'SubRequirementListInstantiation' :
      # allow creation of new subcontractors at any time
      ...
      ENDTASK
    TASK 'SubRequirementListSatisfaction' :
      # continuously wait for all subcontractor components to arrive
      ...
      ENDTASK
    TASK 'ManageSubContractors' :
      # manage business relations with subcontractors
      ...
      ENDTASK
  ENDCONC

  DO 'Integration'

  DO 'Validation'
  CONDITION 'Does the integrated component pass validation?'
  IF 'No' :
    LINK TO 'ReEngineerComponent'
  ENDIF
  IF 'Yes' :
    ENDTIF
  ENDCOND

  INTERACT WITH '$CREATOR.SubRequirementListSatisfaction'
  FOR 'GatherComponents'
}

**Figure 8.5:** Partial Romula representation of the realistic contractor example given in Figure 8.4

A common problem encountered with encoding this contractor was the necessity to model request interactions (namely those in *SendIntermediateComponent* and *SubRequirementListSatisfaction* tasks), within repeatable conditions. In a sense, these looping interactions are a 'fix' for a problem in Romula. A Romula role cannot instinctively 'react' asynchronously to an arbitrary interaction from another role. This problem is demonstrated in Figure 8.6.
Role A can request something at any time and as many times as required from Role B, so Role B must be able to accommodate this eventuality. Ordinarily, expressing this in a RAD could be done using an external event in Role B, but Romula's event-driven functionality is currently not that mature, only allowing users to activate a role's events. Therefore, such an undesired 'fix' in Romula is currently required to model this occurrence properly.

8.4.4.1 Encoding the ManageParentRelations Task

The activities for managing parent relations are given in Figure 8.7. The list of conditions presented correspond to the functionality of the parent contractor management activities given in section 8.4.3.

By manipulating the parent's sub-contractor list and terminating its own sub-contractors, we can remove a contractor and its sub-contractors from the requirements hierarchy. In this scenario, the parent contractor therefore may not be able to complete its requirement list satisfactorily, so in order to do this, it may have to either assign another sub-contractor 'on the fly' for completing the relevant requirements, or complete them itself. Thus, we are dynamically reorganising the project's contractor infrastructure to complete requirements in response to a contractor termination.
**8.4.4.2 Encoding the SubRequirementListInstantiation Task**

As with the simple contractor definition, Romula must implement creation of new sub-contractors differently to the RAD model, but in this case it has been improved to address problem 6 in the previous section. The new concurrent definition shown in Figure 8.8 allows sub-contractors to be created at any time up until integration.

When a new sub-contractor is begun, by the Start command, the unique instance identity is placed in an on-the-fly declared variable which has scope within a role. This reference is used to interact with the sub-contractor to pass a sub-requirements list obtained from Requirements List Analysis. By Add-ing this sub-contractor to a list of all created sub-contractors, we can keep track of and reference those sub-contractors. Effectively, we have modelled a contractor's real-world list of contracts with its sub-contractors.
The simple Romula contractor model does not allow arbitrary creation of sub-contractors, because after it has passed the refined requirement lists to all sub-contractors at the same time, no more may be created. This could prove impractical; a contractor might need to sub-contract a set of requirements to be completed at any time, since some tasks are naturally delegated to sub-contractors later in a project's lifecycle, like documenting a user manual. Incorporating this capability allows the contractor definition to be more reactive in dynamic situations: new sub-contractors can be started simply when they are required, regardless of a preconceived organisation of sub-contractors.

**8.4.4.3 Encoding the SubRequirementListSatisfaction Task**

Figure 8.9 illustrates the encoding of the SubRequirementListSatisfaction task.

This concurrent task exists throughout the entire sub-contractor create-receive component process until all components are received, providing a solution to problem 7 identified within the simple contractor example. However, here Romula suffers the problem of asynchronous requesting interactions; we need to repeat the condition for every interaction with a sub-contractor for receiving a component.
TASK 'SubRequirementListSatisfaction' :
EXPECTLINK FOR 'WaitForComponents'

CONDITION 'Wait for another component to arrive?'

IF 'Yes' :
# collect a component from _one_ in the list of
# existing subcontractors, noting which one
# returns that component, and then remove
# that subcontractor from the list of all
# subcontractors, then repeat condition
SUBINTERACT WITH '$subContractors.*' FOR 'GatherComponents'
TO '$aSubContractor'
REMOVE '$aSubContractor' FROM '$subContractors'
LINK TO 'WaitForComponents'
ENDIF

IF 'No' :
# do nothing, but we are effectively allowing this
# concurrent task to exit later
ENDIF

ENDCOND
ENDTASK

Figure 8.9: Concurrent definition of the SubRequirementListSatisfaction task

Note the additional syntax included within the GatherComponents interaction statement, which is necessary for collecting completed components from individual sub-contractors. It contains a reference to '$subContractors.*'. This is a feature which was added in the later stages of Romula's development which allows a role instance to request an interaction with any concurrent thread within another role instance, as long as the interaction 'reason' is the same. For example, this feature is necessary here since we cannot tell which concurrent thread in the sub-contractor we may be interacting with, since this interaction could occur with either the ManageParentRelations, or SubRequirementListSatisfaction sub-contractor concurrent threads, or at the end of the main sub-contractor definition once the full component has been integrated and validated.

Also note that we are using the specific feature of the Romula SubInteract command (explained in section 4.8.2) which allows us to request an interaction with any of the sub-contractors listed in the $subContractors role instance list. We then simply capture the instance we have interacted with in the $aSubContractor instance list, and eliminate it from the $subContractors list.

8.4.4.4 Encoding the ManageSubContractors Task

The ManageSubContractors task is encoded in Figure 8.10. Here we can see how each of these management functions are implemented.
Since this task deals with all sub-contractors, we need to know exactly which sub-contractor to apply each function to. This is achieved by using much the same method as in the previous section with the SubInteract statement, placing the selection in the variable $aSubContractor. This enables application of a function to a specific sub-contractor.

TASK 'ManageSubContractors' :
   EXPECTLINK FOR 'ContinueIntermediateProcessing'
   
   CONDITION 'Manage Sub Contractors'
   
   IF 'Request Component and Terminate Sub Contractor' :
      SUBINTERACT WITH '$subContractors.SendIntermediateComponent'
         FOR 'IntermediateComponent' TO '$aSubContractor'
      REMOVE '$aSubContractor' FROM '$subContractors'
      LINK TO 'ContinueIntermediateProcessing'
      ENDF
   
   IF 'Terminate Sub Contractor relation' :
      REMOVE '$aSubContractor' FROM '$subContractors'
      LINK TO 'ContinueIntermediateProcessing'
      ENDF
   
   IF 'Request Intermediate Component' :
      SUBINTERACT WITH '$subContractors.SendIntermediateComponent'
         FOR 'IntermediateComponent'
      LINK TO 'ContinueIntermediateProcessing'
      ENDF
   
   IF 'No more Sub Contractor Management' :
      ENDF
   
   ENDCOND
   ENDTASK

Figure 8.10: Concurrent definition of the ManageSubContractors task

After eliminating a sub-contractor from the requirements hierarchy, the contractor has to either assign another sub-contractor 'on the fly' for completing the relevant requirements, or complete them itself. Thus, we are dynamically reorganising the project's contractor infrastructure to complete requirements in response to a contractor termination. However, note that since there is no feature in Romula which allows us to end the execution of another thread pre-emptively, we have to make do with eliminating sub-contractors from a contractor's list of sub-contractors. Since all the contractors are represented in a hierarchy, when a contractor is extracted, that contractor and all its sub-contractors no longer form part of that hierarchy. In effect, therefore, they are removed from the project, which is no longer dependent on that contractor's component for a completed product.
IF 'Pass unfinished Component and Terminate relations' :
    # prepare a makeshift component from the 
    # work carried out so far, and return it
    DO 'Prepare makeshift component'
    INTERACT WITH '$CREATOR.SubRequirementListSatisfaction'
    FOR 'GatherComponents'
ENDIF

Figure 8.11: Representing part of the ManageParentRelations task as an If
statement in a condition

Encoding the ManageParentRelations task is quite similar. To terminate business relations with a parent
contractor pre-emptively, we simply specify the preparation of a makeshift component, and send it to the
parent contractor via the GatherComponents interaction. The parent contractor accepts the component,
(through its SubRequirementListSatisfaction concurrent thread), and the contractor is then free to remove
itself from the project component hierarchy. Figure 8.11 models how a contractor might terminate its
relations with its parent contractor, returning an incomplete component.

8.5 Summary

We have demonstrated how Romula can represent a requirements-oriented framework, (AMMROF), which
is designed to handle a business infrastructure dynamically. The example models in Romula demonstrate
its ability to handle, naively and more realistically, this requirements-oriented approach to engineering
software, realised as a contractor/sub-contractor business process. The process model utilised can be
defined as flexibly or as restricting as desired, and in almost any manner. The only proviso to representing
a process model in this framework is that the model must conform to the process structure given in section
8.2, to maintain the representation of the project node, or requirements/organisation, hierarchy.

The framework itself exhibits several advantages:

- Requirements remain the sole driver for production; emphasis is on fulfilling
  requirements, and validating components with respect to requirements.
- The framework can be utilised in a contractor/sub-contractor based environment, as
demonstrated, or internally within a software development company, where each node
in the project hierarchy could be a project team, internal department, or both...
- ... which demonstrates that the framework is scaleable.
- Fulfilment of a set of requirements from the framework follows a set pattern of
  behaviour which can also be applied to non-software production tasks such as
documentation. Process definitions can be tailored to fit all types of required artifacts, such as differing types of code components, documentation, etc.

- When sub-contracting, the contractor could specify that a sub-contractor's process model must conform to a specific quality standard (e.g. ISO 9001).

- Assuming the representation of a project's framework is kept maintained by all included parties, the state of the whole project can easily be determined at the highest level.

- If a project node hierarchy for a project could be represented, maintained and accessed from the Internet (or company Intranet) by all participating parties, the framework could visually represent a project's completeness in real time.

The last advantage could only be realised at a process level of visualisation, however, if the involved entities have process capability which supported a 'managed' process (c.f.: Capability Maturity Model (CMM), section 2.4). If the involved entities are not at this level of the CMM, then only fulfilment of requirements could be used as an indicator of project completion. However, it is important to realise that this framework has not been validated, only verified, and hence we cannot attest to its suitability when modelling and enacting a real business infrastructure.

Although we have demonstrated that Romula is capable of modelling such a framework, the complexity of this framework has highlighted some inherent problems with using Romula for this task. As with the validation of Romula conducted in chapter 7 with ProcMod, these problems fall into two similar categories of representation and execution in Romula.

Firstly, problems exist when representing AMMROF in Romula. Romula cannot easily represent the tasks associated with managing business relations with parent and sub-contractors, as illustrated in the more realistic process example, since Romula roles cannot react to asynchronous requests from other roles. They have to be handled using loops and conditions which check for incoming requests. This is certainly not elegant, and can prove somewhat confusing semantically when attempting to understand the Romula representation of such a complex framework. Additionally, we cannot properly pre-emptively remove a sub-contractor from the hierarchy when we wish to terminate business relations; the sub-contractor still 'exists' despite the fact that it no longer has any responsibilities within the project framework. Also, of course, we cannot represent the requirements side of the framework, since Romula is simply not equipped to handle such data.

Secondly, problems were encountered when enacting AMMROF using the Romula animator. Not surprisingly, these problems were also encountered when validating Romula using ProcMod. Again, a
large display is required to execute this model because of its complexity; the numerous concurrent threads within a single contractor create many thread windows which clutter the display when executing multiple contractors, which can lead to an effectively unusable model. Lastly, the lack of any model storage or retrieval facility again proved an inconvenience, in this case when verifying the model.

Despite the fact that we have not validated the framework itself, what we have established is that modelling organisations at this level is a complex task, and a framework which attempts to undertake this task must be able to cope with many organisational eventualities. In addition, these eventualities can be represented within a set structure which allows us to categorise differing types of organisational activities and responsibilities. We have also shown that Romula is capable of modelling this framework, but achieving this has proven to be a complex task which results in a complex model. However, despite this complexity, we can execute and verify the model, which demonstrates that although Romula may not be ideal for modelling such a framework, it can still be achieved. Following on from chapter 7, we have again demonstrated that Romula is capable of modelling complex socio-technical processes and executing them, and by doing so, we are further justifying the claim that Romula is a valid process modelling approach.
9 Conclusions

The primary goal of the work undertaken was to develop a valid process modelling approach which exhibits both the modelling power of the process program and the modelling clarity offered by graphical process notations. What we have produced is a process modelling language which has more modelling-based syntax and semantics, instead of the programming-based syntax and semantics common in other modelling languages. This was achieved by investigating and evaluating current methods of process modelling, and devising a novel approach that overcame the programming-based shortfalls of the modelling languages, whilst utilising the best features of a popular modelling notation. Essentially, this involved omitting the low abstraction level and syntactic complexity of modelling languages, and focusing on the good properties of a notational approach, such as approachability, enactability, usability, and understandability. We have developed an associated animation tool which can successfully execute the language and display the model. We have verified the language and this animation tool by producing a pseudo-realistic process model and executing it. We have validated the language and the animation tool for the purpose of process modelling by enacting this pseudo-realistic process model on a simple, individual-based software project. Additionally, we have substantially specified the semantics of the language which allows models which have been represented in this language to be formally specified and reasoned with. We have further validated the applicability of the approach by using it to model a requirements-oriented software development organisational framework.

9.1 Contributions

The contributions of this thesis to the discipline of process modelling and requirements engineering are summarised below.

9.1.1 Process Modelling and Requirements Engineering

The work presented in this thesis and the discussions of issues raised during the duration of this work contribute the following to the area of process modelling:

- A new, validated process modelling language named Romula which encapsulates the features of usability and enactability. It allows process models to be directly defined in terms of process modelling concepts, and utilises a syntax based on those concepts.
A process modelling language which allows models expressed using the eminently popular Role Activity Diagram (RAD) notation to be easily translated into the language.

A process modelling language which overcomes the RAD modelling disadvantage of role instantiation by providing the ability to track multiple instances of an arbitrary number of roles. This feature extends the concept of inter-role interaction, by enabling interactions between complex social orders to be expressed in a simple, minimalistic description format.

A method which allows process models represented in this language to be translated into a formal specification notation. Models translated in this manner can thus be formally reasoned with, and analysed for various properties.

A process modelling language which can additionally be used to model non-process scenarios which are beyond the initial scope of its requirements. This quality has been demonstrated with the modelling of State Transition Diagrams and an Electronic Point of Sale (EPOS) system. This illustrates the inherent flexibility of the language, and the possibility that it can be applied to modelling diverse scenarios to aid in the possible discovery of solutions.

A process modelling language which is able to model and execute process-oriented scenarios as well as role-oriented (socio-technical) scenarios.

A possible extension to both the RAD notation and the Romula process modelling language which exploits the nature of the software development process. This extension enables the existence of materials within the software development process to be modelled in a manner which promotes traceability and understanding of those materials.

A pseudo-realistic process model, ProcMod, which encapsulates the features of non-functional requirement management, simple component-based software implementation, and simple component-based software integration. Within the context of an individual-based software development environment, the model was found to be most useful in verifying and validating the Romula language and animator. This model could feasibly be used to evaluate other process modelling approaches in a similar way.

A requirements-oriented software development framework, AMMROF, modelled using Romula. It illustrates how the handling of requirements within the context of a contractor/sub-contractor organisational model can be expressed in a format which also captures the process which is enacted to fulfil those requirements.
A discussion of the issues of deviating from a process model and the simple realisation that this is a probable possibility. A suggested risk minimisation method of what could be done if deviation occurs is presented, and also highlighted are the dire consequences that can result from a poor choice to deviate.

A discussion of a method by which the SEI’s Capability Maturity Model may be applied to CASE tools. Such a classification scheme would enable software organisations to adopt those tools appropriate to their level of process maturity.

9.1.2 Tools

The animation tool implemented in Perl/Tk has evolved over the duration of its development which can successfully execute and enact Romula process models. This prototype tool runs equally effectively under the Windows 95 and UNIX operating systems, and is theoretically portable to any platform which fully supports the Perl/Tk interpreter. It provides a window-based graphical user interface that enables executing process models to be displayed in a logical, easily discernible format.

9.1.3 Romula and Software Engineering

In the context of software engineering, Romula provides an approach which can be used to assist in decreasing process uncertainty within software organisations. It achieves this by allowing modellers with no prior programming expertise to define process models in terms of process modelling concepts. In overall application terms, the Romula approach was designed to form part of an overall process definition and improvement strategy, such as the CMM. When organisations are attempting to reach level three process maturity, and hence define their process, they can use Romula to fulfil their process definition needs. It has been demonstrated that this can be achieved in a variety of different ways. The abstract nature of Romula, inherited from Role Activity Diagrams, allows models at any level of abstraction to be defined with ease. Romula can also be utilised to form part of a complex process approach, as demonstrated with the requirements-oriented framework in chapter 8.

Those who have experience with Role Activity Diagrams can utilise Romula in the same capacity with minimal difficulty: translation from RADs to Romula is basically a trivial process, and Romula has been specifically targeted as a complement to RADs. RADs provide an intuitive way of presenting process models, and Romula provides the ability to examine the dynamics of process models. By defining processes in Romula, they can be executed to further the understanding of the nature of an organisation’s process, and hence uncover potential deficiencies within that process. This goal can be further realised by
translating Romula models into the CSP formal notation, and applying formal reasoning to the resultant specification. If a similar CSP translation process exists for other process modelling languages, it becomes possible to combine models with differing process modelling origins and execute and analyse the resultant model.

Section 4.8 details two examples of applying Romula to the modelling of scenarios which are not associated with software process modelling. This demonstrates the inherent flexibility of Romula, and it is certainly conceivable that Romula could be applied to modelling other non-software process modelling scenarios. It has already been demonstrated how Romula can model a simple EPOS system, and therefore at least partly assume the rather unexpected, but perhaps not totally surprising, role of client-server simulator. An additional level of flexibility was demonstrated by implementing a model defined in terms of partial ordering in Romula, described in section 4.8.3. Although Romula was not ideally suited to the task, this model demonstrated how Romula can adequately adopt a process-oriented perspective when describing processes.

9.2 Evaluation of the Romula Modelling Approach

The focus of this evaluation is twofold:

1. Does the Romula approach achieve the primary goals detailed in section 1.2?
2. Have the secondary goals of the work as described in section 1.2 been achieved?

Essentially, the Romula approach was intended to address the primary concerns of usability, approachability, and applicability. Ensuring these criteria are met increases the viability of applying Romula to process modelling scenarios to solve process problems.

9.2.1 Fulfilment of Primary Goals

To develop a textual, modelling-based method of process modelling which embodies the understandability of notational methods with the exploitative power of the process modelling language.

The first objective, therefore, is to determine whether the Romula language adopts process modelling concepts over programming language concepts to the degree that it may be classified as a modelling-based approach as opposed to a programming-based approach.
Stating that we are modelling *processes* and not constructing *programs* is inadequate as a response. Close examination of the Romula language definition reveals many constructs that appear to behave very similarly to some programming language constructs. Romula concurrency, conditions and instance lists are examples where this appears to be true. However, the RAD notation formed the conceptual basis for the Romula syntax and semantics, which is a process modelling notation. We have also discussed in section 4.9 that programming-based concepts have only been used where strictly necessary, and where this has been the case, they have been adapted to fit properly within a modelling context, and not directly inserted into the language.

Because Romula is based on a modelling notation, it should follow that the Romula approach is also as understandable as the notational approach on which it is based. Basically, if someone can understand Role Activity Diagrams and the modelling concepts behind that notation, that someone should without too much effort be able to model processes in Romula. Lastly, it has been substantially demonstrated in chapter 4 that Romula is an executable modelling method.

In addition, it is interesting to note that Romula, whilst not being an entire process modelling architecture, at its current state fulfils three of the process definition design issues identified by the Debus-Booch methodology, which were mentioned briefly in section 3.3.6: *step selection* within a process model is catered for by the Romula animator; *control condition selection*, *control flow selection*, and *concurrency specification* are catered for by the Romula *Condition*, *Link/ExpectLink*, and *Concurrent* commands respectively. *Refinement selection* is catered for at a satisfactory level by the *Do* task abstraction, but at its current state of maturity Romula cannot hierarchically modularise tasks within a role.

**To develop an associated animation tool that allows models developed in this modelling language to be executed.**

**and**

**To validate how well this modelling approach could model and enact a pseudo-realistic process model.**

Whether or not Romula has met these goals has already been ascertained and covered in appropriate detail in section 4.6 and chapter 7 respectively, but the results will be summarised briefly for the purpose of completeness.

Verification was a very straightforward process. It simply involved constructing both realistic and unrealistic process models which tested the animator's ability to execute them. Since most verification was conducted during the actual development of the animator, the formal verification process basically
consisted of re-executing these test models in sequence. It was concluded that undoubtedly Romula was capable of executing process models.

Validation proved to be a far more complex process. During the development of Romula, a process model was devised in parallel with the interpreter. The model was created to be pseudo-realistic, in that it incorporated various features commonly found in a waterfall-based software development process, in addition to some more advanced features such as a non-functional requirement gathering and validation activity. A simple calculator project was devised to act as an enaction scenario. In short, it was determined that the Romula animator could effectively enact process models, with two exceptions. Firstly, the effectiveness of the user interface depended greatly on the size of the computer monitor on which it was displayed, and secondly, that there is no model state storage or retrieval facility. These two provisos were identified in section 7.2.4.2 and are addressed in section 9.3.2.

9.2.2 Fulfilment of Secondary Goals

To validate the realism of the pseudo-realistic process model [ProcMod] by enaction using the modelling language and animation tool to ascertain if the validation process is itself valid, and to further validate the modelling language's applicability for process modelling.

As described in section 7.2, a simulated multi-person approach to validating ProcMod was adopted, and a simple software calculator scenario was the basis for enacting ProcMod. In general, it was concluded that the model itself provided a sufficiently realistic foundation to devise and implement the calculator scenario. However, it should be noted that since the model was enacted in an individual-based development environment which simulated a multi-person development environment, the validity of this model with respect to a real development team cannot be assumed.

The results of this validation process were twofold:

- A list of minor alterations that could be made to ProcMod to make it more realistic were described.
- Four possible fundamental approaches to altering ProcMod at a more abstract level were detailed. (See section 7.3.3.2).

Despite the reservations concerning its realism, within the validation context for which it was designed, it performed well and was deemed to be a valid model. This thus adds more evidence that Romula is a valid process modelling approach, since Romula modelled and enacted this process. In addition, as a
consequence of the minor observations made concerning ProcMod, ProcMod was improved from version 5.2 (see appendix B) to version 6 (see appendix C).

To determine how effectively this modelling approach could offer process support for the handling of software requirements in a requirements-oriented context.

Chapter 8 describes a framework for representing the state and progress of a requirements-oriented software development project with multiple contributing entities. The notion was introduced that requirements have an associated process that is enacted to fulfil those requirements, and the proposed framework maintains a consistent focus on requirements as the driver for software production. It was illustrated and consequently demonstrated that this abstract, high-level approach to project management is a viable foundation for modelling this type of development scenario, although proper evaluation of this approach is required. In addition, it was demonstrated that although the resultant model of the framework was complex, Romula could provide adequate support for this modelling and consequent enaction process. This achievement further validates the applicability of Romula to the area of process modelling.

9.3 Future Work

It has been concluded that because of Romula's flexibility and breadth of possible applications, it represents a well-defined, solid foundation for further development.

9.3.1 Improvement of Romula

During the development of the Romula animator, it became apparent that many possible avenues of improvement exist:

- **RAD Visualiser:** The overview of the ASPEN [Doheny and Filby 1996] tool architecture (described in section 6.5.3) includes a RAD visualiser for viewing process models. Also, current work in our department is concerned with how RolEnact [Phalp et al. 1998] models can be visualised as Role Activity Diagrams. Although RolEnact provides an event-driven model of process representation, a tool has been developed which translates these RolEnact models into displayable RADs. A valid possible extension to the Romula animator would be a modified version of this tool, or a proprietary creation, which allows this type of model visualisation. A
window could concurrently display several models in RAD format being executed (or enacted), allowing an alternative method of depicting the state of these models.

- **RAD Modeller:** A more interesting and valuable addition would be a RAD construction front-end, which would enable the modeller to construct a model in the RAD format. It would then become a feasible possibility to build an automated translator which converts the RAD representation into an executable Romula representation. The RAD visualiser could then display the state of the executing (or enacting) process model, with the modeller being totally unaware that Romula is being used to drive the execution or enaction of the model.

- **Romula-to-CSP Translator:** We have discussed in chapter 5 the straightforward process of translation from Romula to CSP. We can further trivialise this process by providing a tool which allows Romula models to be *automatically* translated into CSP specifications.

- **Implementation of Material Modelling:** As introduced and demonstrated in section 4.7, it would be possible to extend the Romula language definition to incorporate the modelling of materials through the development process. By enabling the animator to keep track of material creation, refinement, passing, copying, etc. it could easily be determined at any stage in the process which roles have access to which materials. Additionally, it would be obvious when materials are created, completed, etc.

- **Distributed Enaction:** By enabling separate models on separate computers to be able to interact with each other, it could become feasible for every participating role in a process model enaction to have their own interface which deals with their own process role definition. As an interaction is completed between roles in the real world, the Romula enacting representations modelling this interaction are informed that it is complete, and the enaction of each role then continues separately. This could either be implemented using a client-server model, or using a peer-to-peer model.

- **Storage/Retrieval of Enacting Models:** As mentioned previously in section 7.2.4.2, the lack of a facility to store and retrieve Romula models which are being enacted is a distinct inconvenience. This would prove an essential addition to the Romula animator.

- **Improvement of the Graphical User Interface:** As identified in section 7.2.4.2, the user interface is not ideally suited to displaying the execution of large models on smaller monitors. The display can become cluttered and unclear. By redesigning and re-implementing the original role instance window (see section 4.5.4.2) to be smaller, the screen window clutter would be significantly reduced.
9.3.2 Further Research

Many of the discussions presented in this thesis require much further research:

- **Process Deviance:** More research needs to be conducted to ascertain the nature and consequences of deviance as discussed in section 3.4, and to create appropriate methods for representing task importance in the model, and measuring the 'purity' of an enacted model. It may be then possible to extend the Romula language and animator to handle how this deviance may be measured and calculated for a particular enacting model.

- **Partial Ordering of Tasks:** This method of process modelling as described in section 4.8.3 represents the most fundamental shift in modelling processes. Although this type of method has been seen before (see Interact/Intermediate in section 3.3.3) this represents an opposing style of process modelling (process-oriented models as opposed to role-oriented (socio-technical) models) which was adopted by Romula. Development of a new language which properly encompasses this opposing approach (perhaps called Remus?) would then be a logical step.

- **Further Generalisation and Specification of Romula:** In its present state, the specification of the Romula semantics in CSP is only really partially complete and partially generalised. As discussed in chapter 5, we cannot, at the current time, specify interaction initiation due to a notational conflict. This needs to be further researched. In addition, a much greater understanding of the semantics could be achieved by developing a CSP specification which, at a greater level of generalisation, could describe absolutely any Romula model in more abstract terms.

9.4 Closing Remarks

As the demands for software complexity increase, better, more complex development processes are required to create this software. However, such complexity in both cases can lead to increased product quality concerns which need to be addressed. Process modelling languages can not only offer organisations the process stability and predictability, but also overall employee awareness of that process, and their roles within it. Recent years have seen greatly increasing numbers of process modelling and process improvement techniques enter the field of software engineering, and process modelling languages form an important part of those techniques. However, it is this authors belief that whilst the use of process modelling languages may well increase over the next few years, it is simply the approachability of those languages that will determine their success in industry, and their continued survival.
Appendix A - Romula's Grammatical Syntax

A.1 Low-level Grammar Constructs

Note the distinction between referencing an instance list for storing (\texttt{PUT\_instance\_list}) and for retrieving (\texttt{GET\_instance\_list}). This is necessary since it does not make sense to append to an instance list you are retrieving from, since it would be appending null data. Also note the definition \texttt{GET\_multiple\_ins\_lists} since we can specify an interaction with the role instances from multiple instance lists if required.

\begin{verbatim}
rolename := <alphabetic>
roleid := <rolename> <numeric>
instance_listname := <alphabetic>
GET_instance_list := <instance_listname>
PUT_instance_list := '>' <instance_listname>
GET_multiple_ins_lists := <instance_listname> | <instance_listname> <GET_multiple_ins_lists>
role_ref := <rolename> | <GET_multiple_ins_lists>
remark := '#' <alphanumeric>
\end{verbatim}

A.2 Romula Top-Level Definition

It is important to note that it is not imperative that the \texttt{romula\_file} definition includes any \texttt{START} statements at the beginning of the definition. There is, however, no reason why a user would want to do this. A Romula definition would simply not execute.

\begin{verbatim}
romula_file := <remark> <romula_file> 
    | START <rolename> 
    | <romula_file> 
    | <role_def_list>
\end{verbatim}
A.3 Romula Role Definition

This is straightforward; we can define any natural number of role definitions, each with an arbitrary number of events, if required.

\[
\begin{align*}
\text{role_def_list} & := \text{role_def} \ (\text{role_def_list}) \mid \text{role_def} \\
\text{role_def} & := \text{ROLE} \ <\text{rolename}> \\
& \quad \{ \text{block} \} \\
& \quad \mid \text{ROLE} \ <\text{rolename}> \ \\
& \quad \text{event_def_list} \\
& \quad \{ \text{block} \} \\
\text{event_def_list} & := \text{event} \ (\text{event_def_list}) \mid \text{event} \\
\text{event} & := \text{EVENT} \ <\text{eventname}> \\
& \quad \{ \text{block} \}
\end{align*}
\]

A.4 Romula Role Command Definition

The main Romula role command definition. Note that we have used the specific instance list references GET_instance_list and PUT_instance_list. The definitions of GETINSTANCE and PASSINSTANCE specifically require an instance list role reference for identification purposes. If that instance list contains more than one instance, only the first is used.

\[
\begin{align*}
\text{block} & := \text{<command>} \ (\text{block}) \mid \text{<command>} \\
\text{command} & := \text{START} \ <\text{rolename}> \\
& \quad \mid \text{START} \ <\text{rolename}> \ \text{TO} \ <\text{PUT_instance_list}> \\
& \quad \mid \text{DO} \ <\text{task}> \\
& \quad \mid \text{INTERACT WITH} \ <\text{role_ref}> \ \text{FOR} \ <\text{reason}> \\
& \quad \mid \text{INTERACT WITH} \ <\text{role_ref}> \ \text{FOR} \ <\text{reason}> \ \text{TO} \ <\text{PUT_instance_list}> \\
& \quad \mid \text{SUBINTERACT WITH} \ <\text{role_ref}> \ \text{FOR} \ <\text{reason}> \\
& \quad \mid \text{SUBINTERACT WITH} \ <\text{role_ref}> \ \text{FOR} \ <\text{reason}> \ \text{TO} \ <\text{PUT_instance_list}> \\
& \quad \mid \text{GETINSTANCE FROM} \ <\text{GET_instance_list}> \ \text{FOR} \ <\text{reason}> \ \text{TO} \\
& \quad \mid \text{PASSINSTANCE TO} \ <\text{GET_instance_list}> \ \text{FOR} \ <\text{reason}> \ \text{FROM} \\
& \quad \mid \text{EXPECTLINK FOR} \ <\text{reason}> \\
& \quad \mid \text{LINK TO} \ <\text{reason}> \\
& \quad \mid \text{<concurrent>} \\
& \quad \mid \text{<condition>} \\
& \quad \mid \text{<remark>} \\
& \quad \mid \text{<emptyline>}
\end{align*}
\]
A.5 Romula Higher-Level Control Flow Definitions

Since the \textit{CONCURRENT} and \textit{CONDITION} constructs are complex definitions, they need to be defined separately.

\begin{verbatim}
concurrent := CONCURRENT <task_list> ENDCONC

<task_list> := <task> <task_list> | <task>

<task> := TASK <taskid> :
             <block> ENDTASK

condition := CONDITION <condition_list> ENDCOND

<condition_list> := <if> <condition_list> | <if>

<if> := IF <condition_name> :
             <block> ENDIF
\end{verbatim}
Appendix B: Pseudo-Realistic Process Model
ProcMod v5.2

B.1 RAD Representation of ProcMod v5.2

Figure B.1: Requirements Analysis Phase of ProcMod v5.2
Figure B.2: Design Phase of ProcMod v5.2
Figure B.3: Component Engineering Phase of ProcMod v5.2
Figure B.4: Integration Phase of ProcMod v5.2
B.2 Romula Representation of ProcMod v5.2

# Requirements-oriented Process Model: v5.2

START 'Project_Manager'
START 'Customer'

ROLE 'Project_Manager'
|

# ***** STAGE: Requirements *****

START 'Requirements_Analyst'
START 'Integration_Engineer'

INTERACT WITH 'Requirements_Analyst' FOR 'PassProjectPlan'
INTERACT WITH 'Requirements_Analyst' 'Customer' FOR 'EstablishContact'
EXPECTLINK FOR 'RedoRequirements'
INTERACT WITH 'Requirements_Analyst' 'Customer' FOR 'RequirementsReview'
INTERACT WITH 'Requirements_Analyst' 'Customer' FOR 'PassReqSpec'
INTERACT WITH 'Requirements_Analyst' 'Customer' FOR 'RequirementsSpecReview'

CONDITION 'GROUP: Requirements Spec OK?'
   IF 'Satisfactory' :
   ENDIF
   IF 'Unsatisfactory' :
      LINK TO 'RedoRequirements'
      ENDIF
ENDCOND

# ***** STAGE: Design *****

START 'Designer'

INTERACT WITH 'Designer' FOR 'PassReqDocument'
INTERACT WITH 'Designer' 'Customer' FOR 'PrelimDesignReview'
EXPECTLINK FOR 'RedoDesign'
INTERACT WITH 'Designer' 'Customer' FOR 'DesignReview'

CONDITION 'GROUP: Design OK?'
   IF 'Satisfactory' :
   ENDIF
   IF 'Unsatisfactory' :
      LINK TO 'RedoDesign'
      ENDIF
ENDCOND

INTERACT WITH 'Designer' 'Integration_Engineer' FOR 'PassBuildPlan'

# ***** STAGE: Implementation *****

EXPECTLINK FOR 'StartCompEngineers'
START 'Component_Engineer' TO '>$engineers'
INTERACT WITH 'Component_Engineer' FOR 'PassWorkload'

CONDITION 'Start another Component Engineer?'
   IF 'No' :
   ENDIF
IF 'Yes' :
    LINK TO 'StartCompEngineers'
ENDIF

INTERACT WITH '$engineers' FOR 'CollectReports'
DO 'Prepare Implementation Report'
INTERACT WITH 'Customer' FOR 'ImplementationReview'

# ***** STAGE: Integration and Testing *****
START 'Integration_Engineer'
PASSINSTANCE TO 'Integration_Engineer' FOR 'PassEngineerList' FROM '$engineers'
INTERACT WITH 'Integration_Engineer' FOR 'PassIntegReport'
INTERACT WITH 'Customer' FOR 'IntegrationReview'
INTERACT WITH 'Integration_Engineer' 'Customer' FOR 'ValidationTestReview'
INTERACT WITH 'Integration_Engineer' FOR 'ConfigurationAudit'

ROLE 'Requirements_Analyst'
{
INTERACT WITH 'Project_Manager' FOR 'PassProjectPlan'
INTERACT WITH 'Project_Manager' 'Customer' FOR 'EstablishContact'
EXPECTLINK FOR 'RedoRequirements'
CONCURRENT

TASK 'Modelling' :
    DO 'Select appropriate modelling approaches'
    DO 'Construct system models'
ENDTASK

TASK 'Interviews' :
    INTERACT WITH 'Customer' FOR 'Interviews'
ENDTASK

TASK 'DocumentAnalysis' :
    DO 'Document/Data Analysis'
ENDTASK

TASK 'ProblemPartitioning' :
    DO 'Problem Partitioning'
ENDTASK

ENDCONC

DO 'Represent system model in Telos'
INTERACT WITH 'Project_Manager' 'Customer' FOR 'RequirementsReview'
CONCURRENT

TASK 'Taxonomy' :
    DO 'Construct NFR Taxonomy for new system'
ENDTASK

TASK 'GoalDecomposition' :
    DO 'Select GDM'
DO 'Decompose NFRs into GDM-based goals'
ENDTASK

TASK 'UserManual' :
DO 'Construct User Manual'
ENDTASK

TASK 'ValidCriteria' :
DO 'Construct Validation Criteria'
ENDTASK

ENDCONC

DO 'Produce Requirements Specification document'

INTERACT WITH 'Integration_Engineer' FOR 'PassValidCriteria'

INTERACT WITH 'Project_Manager' 'Customer' FOR 'PassReqSpec'
INTERACT WITH 'Project_Manager' 'Customer' FOR 'RequirementsSpecReview'

CONDITION 'GROUP: Requirements Spec OK?'
IF 'Satisfactory' :
ENDIF
IF 'Unsatisfactory' :
LINK TO 'RedoRequirements'
ENDIF
ENDCOND

ROLE 'Customer'

INTERACT WITH 'Project_Manager' 'Requirements_Analyst' FOR 'EstablishContact'
EXPECTLINK FOR 'RedoRequirements'

INTERACT WITH 'Requirements_Analyst.Interviews' FOR 'Interviews'

INTERACT WITH 'Project_Manager' 'Requirements_Analyst' FOR 'RequirementsReview'
INTERACT WITH 'Project_Manager' 'Requirements_Analyst' FOR 'PassReqSpec'
INTERACT WITH 'Project_Manager' 'Requirements_Analyst' FOR 'RequirementsSpecReview'

CONDITION 'GROUP: Requirements Spec OK?'
IF 'Satisfactory' :
ENDIF
IF 'Unsatisfactory' :
LINK TO 'RedoRequirements'
ENDIF
ENDCOND

INTERACT WITH 'Project_Manager' 'Designer' FOR 'PrelimDesignReview'
EXPECTLINK FOR 'RedoDesign'

INTERACT WITH 'Project_Manager' 'Designer' FOR 'DesignReview'

CONDITION 'GROUP: Design OK?'
IF 'Satisfactory' :
ENDIF
IF 'Unsatisfactory' :
LINK TO 'RedoDesign'
ENDIF
ENDCOND

INTERACT WITH 'Project_Manager' FOR 'ImplementationReview'

INTERACT WITH 'Project_Manager' FOR 'IntegrationReview'
INTERACT WITH 'Project_Manager' 'Integration_Engineer' FOR 'ValidationTestReview'

ROLE 'Designer'

INTERACT WITH 'Project_Manager' FOR 'PassReqDocument'

DO 'Translate Telos into TDL'

CONCURRENT

TASK 'DataDesign' :
  DO 'Translate Requirements and TDL into Data Design'
  ENDTASK

TASK 'SoftwareArchDesign' :
  DO 'Translate Requirements and TDL into Software Architecture'
  ENDTASK

TASK 'GoalSatisficing' :
  DO 'Select GSM'
  DO 'Ensure NFRs are satisficed throughout Design'
  ENDTASK

ENDCONC

INTERACT WITH 'Project_Manager' 'Customer' FOR 'PrelimDesignReview'

EXPECTLINK FOR 'RedoDesign'

CONCURRENT

TASK 'RefineArch' :
  DO 'Refine S/W Arch & Data Design into lower-level modules'
  DO 'Translate into Pseudocode'
  ENDTASK

TASK 'InterfaceDesign' :
  DO 'Interface Design'
  ENDTASK

TASK 'CreateTestPlan' :
  DO 'Create Test Plan'
  ENDTASK

ENDCONC

DO 'Internal Technical Review'

CONDITION 'Design OK?'
  IF 'Satisfactory' :
    ENDF
  IF 'Unsatisfactory' :
    LINK TO 'RedoDesign'
    ENDF
ENDCOND

DO 'Create Build Plan'

INTERACT WITH 'Integration_Engineer' FOR 'PassTestPlan'

INTERACT WITH 'Project_Manager' 'Customer' FOR 'DesignReview'

CONDITION 'GROUP: Design OK?'

182
IF 'Satisfactory' :
    ENDIF
IF 'Unsatisfactory' :
    LINK TO 'RedoDesign'
    ENDIF
ENDCOND

INTERACT WITH 'Project_Manager' 'Integration_Engineer' FOR 'PassBuildPlan'

)}

ROLE 'Component_Engineer'
{
INTERACT WITH '$CREATOR' FOR 'PassWorkload'
EXPECTLINK FOR 'DoNextModule'
DO 'Prepare next module workload'
EXPECTLINK FOR 'ReimplementModule'
DO 'Implement module'
DO 'Blackbox test module'
DO 'Whitebox test module'
CONDITION 'Module pass tests?'
    IF 'Yes' :
        ENDIF
    IF 'No' :
        LINK TO 'ReimplementModule'
        ENDIF
ENDCOND
CONDITION 'Completed all modules?'
    IF 'Yes' :
        ENDIF
    IF 'No' :
        LINK TO 'DoNextModule'
        ENDIF
ENDCOND

DO 'Prepare Implementation Report'
INTERACT WITH '$CREATOR' FOR 'CollectReports'
INTERACT WITH 'Integration_Engineer' FOR 'CollectModules'

}

ROLE 'Integration_Engineer'
{
EXPECTLINK FOR 'RedoRequirements'
INTERACT WITH 'Requirements_Analyst' FOR 'PassValidCriteria'
CONDITION 'GROUP: Requirements Spec OK?'
    IF 'Satisfactory' :
        ENDIF
    IF 'Unsatisfactory' :
        LINK TO 'RedoRequirements'
        ENDIF
ENDCOND

EXPECTLINK FOR 'RedoDesign'
INTERACT WITH 'Designer' FOR 'PassTestPlan'

183
CONDITION 'Design OK?'
   IF 'Satisfactory'
      ENDIF
   IF 'Unsatisfactory'
      LINK TO 'RedoDesign'
      ENDIF
ENDCOND

INTERACT WITH 'Project_Manager' 'Designer' FOR 'PassBuildPlan'

GETINSTANCE FROM 'Project_Manager' 'PassEngineerList' TO '$engineers'

INTERACT WITH '$engineers' FOR 'CollectModules'

EXPECTLINK FOR 'TestNextCluster'

DO 'Prepare Integration Plan for Module Cluster'

EXPECTLINK FOR 'RetestModule'

DO 'Integrate Module Cluster'

DO 'Test Module Cluster wrt Test Plan'

CONDITION 'Module Cluster passed tests?'
   IF 'Yes'
      ENDIF
   IF 'No'
      START 'Component_Engineer' TO '$debugger'
         INTERACT WITH '$debugger' FOR 'PassWorkload'
         INTERACT WITH '$debugger' FOR 'CollectReports'
         INTERACT WITH '$debugger' FOR 'CollectModules'
      ENDIF
      LINK TO 'RetestModule'
   ENDIF
ENDCOND

CONDITION 'All module clusters integrated?'
   IF 'No'
      LINK TO 'TestNextCluster'
      ENDIF
   IF 'Yes'
      ENDIF
ENDCOND

DO 'Prepare Integration Report'

INTERACT WITH 'Project_Manager' FOR 'PassIntegReport'

EXPECTLINK FOR 'NextValidTest'

DO 'Test next Validation Test wrt Validation Test Criteria'

CONDITION 'Validation fault?'
   IF 'No'
      DO 'Mark test as successful'
      ENDIF
   IF 'Yes'
      DO 'Document fault'
      DO 'Mark test as failed'
      ENDIF
ENDCOND

CONDITION 'Validation Testing complete?'
   IF 'No'
      LINK TO 'TestNextModule'
   ENDIF
ENDIF

IF 'Yes' :
    ENDF

ENDCOND

INTERACT WITH 'Project_Manager' 'Customer' FOR 'ValidationTestReview'
INTERACT WITH 'Project_Manager' FOR 'ConfigurationAudit'

}
Appendix C: Pseudo-Realistic Process Model

ProcMod v6.0

C.1 RAD Representation of ProcMod v6.0

Figure C.1: Requirements Analysis Phase of ProcMod v6.0
Figure C.2: Design Phase of ProcMod v6.0
Figure C.3: Component Engineering Phase of ProcMod v6.0
Figure C.4: Integration Phase of ProcMod v6.0
Appendix D - Data Flow Representation of ProcMod's Deliverables

D.1 DFD Representation of ProcMod v6.0

Figure D.1: Requirements Phase DFD

Figure D.2: Design Phase DFD

Figure D.3: Component Engineering Phase DFD

Figure D.4: Integration Phase DFD
E.1 Romula Representation of the Requirements Phase

DFD

# Requirements
ROLE 'DocumentData_Analysis'
{
    DO 'Document Analysis'
    DO 'Data Analysis'

    INTERACT WITH 'System_Models', 'Functional_Requirements' FOR 'DocData-SysMod'
    INTERACT WITH 'Functional_Requirements' FOR 'DocData-FuncReqs'
    INTERACT WITH 'Problem_Partitioning' FOR 'DocData-ProbPart'
}

ROLE 'Interviews'
{
    DO 'Interviews with Customer'

    INTERACT WITH 'System_Models' FOR 'Interview-SysMod'
    INTERACT WITH 'Problem_Partitioning' FOR 'Interview-ProbPart'
    INTERACT WITH 'NFR_Taxonomy' FOR 'Interview-NFRTax'
    INTERACT WITH 'Functional_Requirements' FOR 'Interview-FuncReqs'
}

ROLE 'System_Models'
{
    INTERACT WITH 'DocumentData_Analysis' FOR 'DocData-SysMod'
    INTERACT WITH 'Interviews' FOR 'Interview-SysMod'

    DO 'Construct System Models'

    INTERACT WITH 'Problem_Partitioning' FOR 'SysMod-ProbPart'
    INTERACT WITH 'Preliminary_UserManual' FOR 'SysMod-Prelim'
    INTERACT WITH 'Requirements_Document' FOR 'SysMod-ReqDoc'
    INTERACT WITH 'Functional_Requirements' FOR 'SysMod-FuncReqs'
}

ROLE 'Problem_Partitioning'
{
    INTERACT WITH 'System_Models' FOR 'SysMod-ProbPart'
    INTERACT WITH 'DocumentData_Analysis' FOR 'DocData-ProbPart'
    INTERACT WITH 'Interviews' FOR 'Interview-ProbPart'

    DO 'Problem Partitioning'

    INTERACT WITH 'Functional_Requirements' FOR 'ProbPart-FuncReqs'
    INTERACT WITH 'Requirements_Document' FOR 'ProbPart-ReqDoc'
}

ROLE 'NFR_Taxonomy'
{
    INTERACT WITH 'Interviews' FOR 'Interview-NFRTax'
}
DO 'Construct NFR Taxonomy'

INTERACT WITH 'NFRGoal_Decomposition' FOR 'NFRTax-NFRGoal'

}

ROLE 'Preliminary_UserManual'
{
    INTERACT WITH 'System_Models' FOR 'SysMod-Prelim'
    INTERACT WITH 'Functional_Requirements' FOR 'FuncReqs-Prelim'

    DO 'Construct Preliminary User Manual'

    INTERACT WITH 'Requirements_Document' 'Prelim-ReqDoc'
}

ROLE 'Functional_Requirements'
{
    INTERACT WITH 'System_Models' FOR 'SysMod-FuncReqs'
    INTERACT WITH 'DocumentData_Analysis' FOR 'DocData-FuncReqs'
    INTERACT WITH 'Interviews' FOR 'Interview-FuncReqs'
    INTERACT WITH 'Problem_Partitioning' FOR 'ProbPart-FuncReqs'

    DO 'List Functional Requirements'

    INTERACT WITH 'Preliminary_UserManual' FOR 'FuncReqs-Prelim'
    INTERACT WITH 'Requirements_Document' FOR 'FuncReqs-ReqDoc'
}

ROLE 'NFRGoal_Decomposition'
{
    INTERACT WITH 'NFR_Taxonomy' FOR 'NFRTax-NFRGoal'

    DO 'Non-Functional Requirements Decomposition'

    INTERACT WITH 'Requirements_Document' FOR 'NFRGoal-ReqDoc'
    INTERACT WITH 'Validation_TestCriteria' FOR 'NFRGoal-ValidTest'
}

ROLE 'Validation_TestCriteria'
{
    INTERACT WITH 'NFRGoal_Decomposition' FOR 'NFRGoal-ValidTest'

    DO 'Construct Validation Test Criteria'

    INTERACT WITH 'Requirements_Document' FOR 'ValidTest-ReqDoc'
}

ROLE 'Requirements_Document'
{
    INTERACT WITH 'System_Models' FOR 'SysMod-ReqDoc'
    INTERACT WITH 'Functional_Requirements' FOR 'FuncReqs-ReqDoc'
    INTERACT WITH 'Problem_Partitioning' FOR 'ProbPart-ReqDoc'
    INTERACT WITH 'Preliminary_UserManual' FOR 'Prelim-ReqDoc'
    INTERACT WITH 'NFRGoal_Decomposition' FOR 'NFRGoal-ReqDoc'
    INTERACT WITH 'Validation_TestCriteria' FOR 'NFRGoal-ReqDoc'

    DO 'Construct Requirements Document'
}
E.2 Romula Representation of the Design Phase DFD

```
# Design
ROLE 'System_Models'
{
    INTERACT WITH 'Design_Model' FOR 'SysMod-DesMod'
}

ROLE 'Functional_Requirements'
{
    INTERACT WITH 'Design_Model' FOR 'FuncReqs-DesMod'
    INTERACT WITH 'Software_Architecture' FOR 'FuncReqs-SoftArch'
    INTERACT WITH 'Data_Design' FOR 'FuncReqs-DataDes'
}

ROLE 'NFRGoal_Decomposition'
{
    INTERACT WITH 'Software_Architecture' FOR 'NFRGoal-SoftArch'
    INTERACT WITH 'Data_Design' FOR 'NFRGoal-DataDes'
}

ROLE 'Design_Model'
{
    INTERACT WITH 'System_Models' FOR 'SysMod-DesMod'
    INTERACT WITH 'Functional_Requirements' FOR 'FuncReqs-DesMod'
    DO 'Design Model'

    INTERACT WITH 'Data_Design' FOR 'DesMod-DataDes'
    INTERACT WITH 'Software_Architecture' FOR 'DesMod-SoftArch'
    INTERACT WITH 'Design_Document' FOR 'DesMod-DesDoc'
}

ROLE 'Data_Design'
{
    INTERACT WITH 'Functional_Requirements' FOR 'FuncReqs-DataDes'
    INTERACT WITH 'NFRGoal_Decomposition' FOR 'NFRGoal-DataDes'
    INTERACT WITH 'Design_Model' FOR 'DesMod-DataDes'
    DO 'Data Design'

    INTERACT WITH 'Software_Architecture' FOR 'DataDes-SoftArch'
    INTERACT WITH 'Modular_Design' FOR 'DataDes-ModDes'
    INTERACT WITH 'Design_Document' FOR 'DataDes-DesDoc'
    INTERACT WITH 'Pseudocode' FOR 'DataDes-PseudoCode'
}

ROLE 'Software_Architecture'
{
    INTERACT WITH 'Design_Model' FOR 'DesMod-SoftArch'
    INTERACT WITH 'Functional_Requirements' FOR 'FuncReqs-SoftArch'
    INTERACT WITH 'NFRGoal_Decomposition' FOR 'NFRGoal-SoftArch'
    DO 'Software Architecture'

    INTERACT WITH 'Modular_Design' FOR 'SoftArch-ModDes'
    INTERACT WITH 'Design_Document' FOR 'SoftArch-DesDoc'
    INTERACT WITH 'Interface_Design' FOR 'SoftArch-IntDes'
}

ROLE 'Modular_Design'
{
    INTERACT WITH 'Software_Architecture' FOR 'SoftArch-ModDes'
    INTERACT WITH 'Data_Design' FOR 'DataDes-ModDes'
    DO 'Modular Design'
```

193
INTERACT WITH 'Build_Plan' FOR 'ModDes-BuildPlan'
INTERACT WITH 'Pseudocode' FOR 'ModDes-PseudoCode'
INTERACT WITH 'Test_Plan' FOR 'ModDes-TestPlan'
INTERACT WITH 'Interface_Design' FOR 'ModDes-IntDes'
INTERACT WITH 'Design_Document' FOR 'ModDes-DesDoc'

ROLE 'Build_Plan'
{
    INTERACT WITH 'Modular_Design' FOR 'ModDes-BuildPlan'
    INTERACT WITH 'Pseudocode' FOR 'PseudoCode-BuildPlan'
    INTERACT WITH 'Interface_Design' FOR 'IntDes-BuildPlan'

    DO 'Build Plan'
}

ROLE 'Pseudocode'
{
    INTERACT WITH 'Modular_Design' FOR 'ModDes-PseudoCode'
    INTERACT WITH 'Data_Design' FOR 'DataDes-PseudoCode'

    DO 'Pseudocode'

    INTERACT WITH 'Build_Plan' FOR 'PseudoCode-BuildPlan'
    INTERACT WITH 'Test_Plan' FOR 'PseudoCode-TestPlan'
}

ROLE 'Interface_Design'
{
    INTERACT WITH 'Software_Architecture' FOR 'SoftArch-IntDes'
    INTERACT WITH 'Modular_Design' FOR 'ModDes-IntDes'

    DO 'Interface Design'

    INTERACT WITH 'Build_Plan' FOR 'IntDes-BuildPlan'
    INTERACT WITH 'Test_Plan' FOR 'IntDes-TestPlan'
    INTERACT WITH 'Design_Document' FOR 'IntDes-DesDoc'
}

ROLE 'Test_Plan'
{
    INTERACT WITH 'Modular_Design' FOR 'ModDes-TestPlan'
    INTERACT WITH 'Pseudocode' FOR 'PseudoCode-TestPlan'

    DO 'Test Plan'

    INTERACT WITH 'Design_Document' FOR 'TestPlan-DesDoc'
}

ROLE 'Design_Document'
{
    INTERACT WITH 'Design_Model' FOR 'DesMod-DesDoc'
    INTERACT WITH 'Modular_Design' FOR 'ModDes-DesDoc'
    INTERACT WITH 'Data_Design' FOR 'DataDes-DesDoc'
    INTERACT WITH 'Software_Architecture' FOR 'SoftArch-DesDoc'
    INTERACT WITH 'Interface_Design' FOR 'IntDes-DesDoc'
    INTERACT WITH 'Pseudocode' FOR 'PseudoCode-DesDoc'
    INTERACT WITH 'Test_Plan' FOR 'TestPlan-DesDoc'

    DO 'Design Document'
}
E.3 Romula Representation of the Component Engineering DFD

```plaintext
# Component Engineering
ROLE 'Data_Design'
{
    INTERACT WITH 'Modules' FOR 'DataDes-Mod'
}

ROLE 'Modular_Design'
{
    INTERACT WITH 'Modules' FOR 'ModDes-Mod'
}

ROLE 'Interface_Design'
{
    INTERACT WITH 'Modules' FOR 'IntDes-Mod'
}

ROLE 'Build_Plan'
{
    INTERACT WITH 'Modules' FOR 'BuildPlan-Mod'
}

ROLE 'Pseudocode'
{
    INTERACT WITH 'Modules' FOR 'PseudoCode-Mod'
}

ROLE 'Modules'
{
    INTERACT WITH 'Data_Design' FOR 'DataDes-Mod'
    INTERACT WITH 'Modular_Design' FOR 'ModDes-Mod'
    INTERACT WITH 'Interface_Design' FOR 'IntDes-Mod'
    INTERACT WITH 'Build_Plan' FOR 'BuildPlan-Mod'
    INTERACT WITH 'Pseudocode' FOR 'PseudoCode-Mod'
    DO 'Implement_Modules'
    INTERACT WITH 'Implementation_Report' FOR 'Mod-ImpRep'
}

ROLE 'Implementation_Report'
{
    INTERACT WITH 'Implementation_Report' FOR 'Mod-ImpRep'
    DO 'Implementation_Report'
}
```

195
E.4 Romula Representation of the Integration Phase DFD

```plaintext
# Integration
ROLE 'Validation_TestCriteria'
{
    INTERACT WITH 'Integrated_System' FOR 'ValidTest-IntegSys'
}

ROLE 'Modules'
{
    INTERACT WITH 'Integrated_System' FOR 'Mod-IntegSys'
}

ROLE 'Test_Plan'
{
    INTERACT WITH 'Integrated_System' FOR 'TestPlan-IntegSys'
}

ROLE 'Integrated_System'
{
    INTERACT WITH 'Validation_TestCriteria' FOR 'ValidTest-IntegSys'
    INTERACT WITH 'Modules' FOR 'Mod-IntegSys'
    INTERACT WITH 'Test_Plan' FOR 'TestPlan-IntegSys'

    DO 'Integrated System'

    INTERACT WITH 'Integration_FailuresDocument' FOR 'IntegSys-IntegFail'
    INTERACT WITH 'Integration_Report' FOR 'IntegSys-IntegRep'
}

ROLE 'Integration_FailuresDocument'
{
    INTERACT WITH 'Integrated_System' FOR 'IntegSys-IntegFail'

    DO 'Integration Failures Document'

    INTERACT WITH 'Integration_Report' FOR 'IntegFail-IntegRep'
}

ROLE 'Integration_Report'
{
    INTERACT WITH 'Integrated_System' FOR 'IntegSys-IntegRep'
    INTERACT WITH 'Integration_FailuresDocument' FOR 'IntegFail-IntegRep'

    DO 'Integration Report'
}```
Appendix F - Romula Representation of the Realistic Hierarchical Process Definition

It is important to realise that although this Romula representation includes all the necessary mechanics of the realistic hierarchical process definition, (with respect to Figure 8.4 in chapter 8), it is not an executable definition. Just inserting this at the beginning of the file would not work:

```
# begin an initial contractor
START 'Contractor'
```

Because the initial role instance of Contractor (the top-level contractor) would have no parent, the first and last Interact commands (effectively simulating the passing of the initial requirements list and the return of the completed product) would not function and the model would deadlock. This would occur since an attempt would effectively be made to initiate an interaction with a null parent (i.e. $CREATOR$ would be undefined). To remedy this situation, an additional role would be required which creates the first Contractor and caters for these interactions (which we could model as the Customer). Such a definition would be similar in approach to the Contractor, although it would not include the first and last Interacts, and the ManageParentRelations, SendIntermediateComponent and ProcessMethod methods would not be required. This structure would allow the Customer to create as many contractors as required for as many products as required.

```
ROLE 'Contractor'
{
    INTERACT WITH "$CREATOR.SubRequirementListInstantiation" FOR 'RequirementsList'
    EXPECTLINK FOR 'ReEngineerComponent'

    CONCURRENT
    TASK 'ManageParentRelations':
        CONDITION 'Manage relations with parent Contractor'
        IF 'Terminate relations':
            # we do not wish to conduct business with the parent
            # contractor any more, so terminate business relations

            # return a 'null' component to the parent contractor
            SUBINTERACT WITH "$CREATOR.SubRequirementListSatisfaction"
                FOR 'GatherComponents'

            # at this point, this contractor and this contractor's
            # sub-contractors must end their part in the framework,
            # but we cannot just pre-emptively 'kill' them, since
            # they then would not be able to terminate relations
            # with their sub-contractors. They must do this themselves.
            ENDIF

        IF 'Pass unfinished Component and Terminate relations':
            # prepare a makeshift component from the
            # work carried out so far, and return it
            DO 'Prepare makeshift component'
            INTERACT WITH "$CREATOR.SubRequirementListSatisfaction"
                FOR 'GatherComponents'
```

197
# sub-contractors must end their part in the framework
# themselves...
ENDIF

IF 'Never prematurely Terminate Parent Contractor relations' :
  # do nothing, but we are effectively allowing
  # the contractor to move on to integration (once
  # all other concurrent tasks have completed)
ENDIF

ENDCOND
ENDTASK

TASK 'SendIntermediateComponent' :
  EXPECTLINK FOR 'SendIntermediateComponent'
  CONDITION 'Send intermediate component to parent contractor?'
  IF 'Yes' :
    INTERACT WITH '$CREATOR.ManageSubContractors'
      FOR 'IntermediateComponent'
    ENDIF
    LINK TO 'SendIntermediateComponent'
  IF 'No more' :
  ENDIF
ENDCOND
ENDTASK

TASK 'ProcessMethod' :
  # do simple process method for contractor's component
  DO 'Design'
  DO 'Implementation'
ENDTASK

TASK 'SubRequirementListInstantiation' :
  EXPECTLINK FOR 'CreateANewSubContractor'
  CONDITION 'Create New Sub-Contractor?'
  IF 'CreateSubContractor' :
    START 'Contractor' TO '$aSubContractor'
    INTERACT WITH '$aSubContractor' FOR 'RequirementsList'
      ADD '$aSubContractor' TO '$subContractors'
    ENDIF
    LINK TO 'CreateANewSubContractor'
  IF 'CreateNoMore' :
    # loop until no more subcontractors are
    # required. A 'null' endif just exits
    # the condition
  ENDIF
ENDCOND
ENDTASK

TASK 'SubRequirementListSatisfaction' :
  EXPECTLINK FOR 'WaitForComponents'
  CONDITION 'Wait for another component to arrive?'
  IF 'Yes' :
    # collect a component from _one_ in the list of
    # existing subcontractors, noting which one
    # returns that component, and then remove
    # that subcontractor from the list of all
    # subcontractors, then repeat condition
    SUBINTERACT WITH '$subContractors.'
      FOR 'GatherComponents' TO '$aSubContractor'
        REMOVE '$aSubContractor' FROM '$subContractors'
      ENDIF
      LINK TO 'WaitForComponents'
  IF 'No' :
    # do nothing, but we are effectively allowing this
    # concurrent task to exit later
  ENDIF
ENDCOND
ENDTASK

TASK 'ManageSubContractors' :
  EXPECTLINK FOR 'ContinueIntermediateProcessing'

CONDITION 'Manage Sub Contractors'
  IF 'Request Component and Terminate Sub Contractor' :
    SUBINTERACT WITH '$subContractors.SendIntermediateComponent'
    FOR 'IntermediateComponent' TO '$aSubContractor'
    REMOVE '$aSubContractor' FROM '$subContractors'
    LINK TO 'ContinueIntermediateProcessing'
  ENDIF
  IF 'Terminate Sub Contractor relation' :
    REMOVE '$aSubContractor' FROM '$subContractors'
    LINK TO 'ContinueIntermediateProcessing'
  ENDIF
  IF 'Request Intermediate Component' :
    SUBINTERACT WITH '$subContractors.SendIntermediateComponent'
    FOR 'IntermediateComponent'
    LINK TO 'ContinueIntermediateProcessing'
  ENDIF
  IF 'No more Sub Contractor Management' :
    ENDFILE
ENDCOND
ENDTASK

ENDCONC

DO 'Integration'

DO 'Validation'
CONDITION 'Does the integrated component pass validation?'
  IF 'No' :
    LINK TO 'ReEngineerComponent'
  ENDIF
  IF 'Yes' :
    ENDFILE
ENDCOND

INTERACT WITH '$CREATOR.SubRequirementListSatisfaction' FOR 'GatherComponents'
}
Bibliography


Henderson, P., 1999. Discussion on how software requirements are represented in industry projects, 28 April 1999.


