Loughborough University

This item was submitted to Loughborough's Research Repository by the author.
Items in Figshare are protected by copyright, with all rights reserved, unless otherwise indicated.

Animation prototyping of formal specifications

PLEASE CITE THE PUBLISHED VERSION

PUBLISHER

© T.S. Hughes

PUBLISHER STATEMENT

This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 2.5 Generic (CC BY-NC-ND 2.5) licence. Full details of this licence are available at: http://creativecommons.org/licenses/by-nc-nd/2.5/

LICENCE

CC BY-NC-ND 2.5

REPOSITORY RECORD

Hughes, Thomas S.. 2019. “Animation Prototyping of Formal Specifications”. figshare. https://hdl.handle.net/2134/27241.

Loughborough University

This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

creative commons

COMMONS DEED

Attribution-NonCommercial-NoDerivs 2.5

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Noncommercial. You may not use this work for commercial purposes.

No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

Disclaimer

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/

BLDSC no:- DX 173557

LOUGHBOROUGH
UNIVERSITY OF TECHNOLOGY
LIBRARY

AUTHOR/FILING TITLE

HUGHES, T.S.

ACCESSION/COPY NO.

040060543

VOL. NO.	CLASS MARK
28 JUN 1996	LOAN COPY

0400605430

BADMINTON PRESS
18 THE HALFCROFT
SYSTON
LEICESTER LE7 8LD
ENGLAND
TEL: 0533 602917
FAX: 0533 696636

Animation Prototyping of Formal Specifications

by

T.S.Hughes

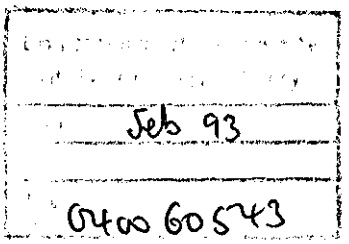
A doctoral thesis

submitted in partial fulfilment of the requirements
for the award of

Doctor of Philosophy of the Loughborough University of Technology

(1992)

© by T.S.Hughes (1992)


Feb 93
0400 60543
W9923918

ANIMATION PROTOTYPING OF FORMAL SPECIFICATIONS.

ABSTRACT.

At the present time one of the key issues relating to the design of real-time systems is the specification of software requirements. It is now clear that specification correctness is an essential factor for the design and implementation of high quality software. As a result considerable emphasis is placed on producing specifications which are not only correct, but provably so. This has lead to the application of mathematically-based formal specification techniques in the software life-cycle model. Unfortunately, experience in safety-critical systems has shown that specification correctness is not, in itself, sufficient. Such specifications must also be comprehensible to all involved in the system development. The topic of this thesis - Animation Prototyping - is a methodology devised to make such specifications understandable and usable. Its primary objective is to demonstrate key properties of formal specifications to non-software specialists. This it does through the use of computer-animated pictures which respond to the dictates of the formal specification.

The major issues discussed in this thesis are:

* The role and presentation of specifications in the development of software for real-time systems.
* The applicability of software prototyping, in particular rapid prototyping, to specification techniques.
* The basis of formal specification techniques and their use as a software specification method.
* The development of a new specification methodology based on a structured application of the Vienna Development Method of software design (defined here as S-VDM).
* The development and construction of automated software tools to enable S-VDM specifications to control the behaviour of graphical displays ("animating the specification").
* The application of S-VDM for the animation of demonstration systems.

DEDICATION.

To Josephine.

ACKNOWLEDGEMENTS.

The author wishes to thank Dr. Jim Cooling for his dedicated supervision of this work. Without Jim's friendship, patience and understanding this thesis would never have been published. Also the author wishes to thank all the members of the Department of Electronic and Electrical Engineering at Loughborough University for their support, in particular John Rippon and Tim Baseley for their assistance with computing matters. Alan Cuff and his colleagues at Transmitton Limited gave very useful comments and encouragement during early trials of animation prototyping presentations. Finally, on a personal level, thank you to Josephine, my wife, for sustaining me to the end in so many ways.

The generosity of Rolls Royce and Associates of Derby in their support of this work is gratefully acknowledged. Also, the author wishes to thank CACI for the provision of the SIMSCRIPT II.5 package which underpins so much of this work. Finally, but most importantly, thanks to the Science and Engineering Research Council without whose financial support none of this work would have been possible.

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS.

$\neg$ Logical Not. The negation of a sentence can be thought of as the logical opposite of that sentence.

$\wedge$ Logical And. The sentence $P \wedge Q$ is referred to the conjunction of P and Q. P and Q are conjuncts of the sentence.

$\vee$ Logical Or. The sentence $P \vee Q$ is referred to as the disjunction of P and Q. P and Q are disjuncts of the sentence. The truth value of the sentence is "P or Q or both P and Q". It is also referred to as inclusive or.

$\Rightarrow$ Logical Implication or "if ... then ... ". The sentence $P \Rightarrow Q$ is read as P implies Q. Care is needed in the informal interpretation of this connective as, in programming terms, there is no information covering "otherwise" or "else". Implication should always be referred to truth tables for exact interpretation.

$\Leftrightarrow$ Logical Equals. This connective is also referred to as double implication. The sentence $P \Leftrightarrow Q$ is true if and only if P and Q have the same truth value.

$\forall$ The universal quantifier. Universal quantification is used to express propositions of the form "every object has this property" or "all objects are related in this way", e.g.

$$\forall x \cdot P(x)$$

$\exists$ The existential quantifier. Existential quantification is used to express propositions of the form "there is at least one object which has this property", e.g.

$$\exists x \cdot P(x)$$

$\vdash$ The syntactic turnstile. If there is a derivation from P to W one can write $P \vdash W$.

$\models$ The semantic turnstile. The statement $P \models W$ can be paraphrased as, given a set of assumptions P are true then W is also true.

$$\frac{P}{C}$$

An Inference rule. The meaning of this is that if all the sentences, P, above the line are given then the sentence below the line, C, can be deduced as an immediate consequence.

$\mathbf{B}$ The set of boolean values, i.e. { true, false }.

$\mathbf{Z}$ The set of integer numbers.

$\mathbf{N}$ The set of natural numbers, positive integers (including zero).

$\mathbf{N_1}$ The set of natural numbers, strictly positive integers (excluding zero).

$\mathbf{R}$ The set of real numbers.

$\in$ Set membership, e.g. $x \in \mathbf{Z}$ denotes that x is a member of the set of integers.

{..} A set of values.

$\{ 1, 2, 4 \}$

Set enumeration.

$\{ n \in \mathbf{N} \mid 1 \leq n \leq 5 \}$

Set comprehension.

$\mapsto$ The mapping operator, e.g. $s \mapsto 123$ denotes that s "maps to" 123.

$\{1 \mapsto x_1, 3 \mapsto x_4\}$

Map enumeration.

$\{ x \mapsto x^2 \in \mathbf{N} \times \mathbf{N} \mid x \in \{ i \in \mathbf{N} \mid -2 \leq i \leq 3 \} \}$

Map comprehension.

[..] A sequence of values.

[12,14,93]

Sequence enumeration.

$doubles = [x \in \mathbf{N} \mid doubles(x) = 2 \times x]$

Sequence comprehension.

T^* Sequence comprehension. The member of the sequence are drawn from the type T.

$\frown$ Sequence concatenation, e.g. $[1,2] \frown [3,4] = [1,2,3,4]$.

$V:T$ Type definition. Variable V is of type T.

$\triangleq$ "Is defined as".

$D \rightarrow R$ Function mapping from domain D to range R.

$A \times B$ The cartesian product of A and B.

$\{P\}Q\{R\}$

A triple. Where P and R are truth valued expressions and Q is some operation. This notation asserts that if the state satisfies expression P then the application of operation Q will yield a state which satisfies the expression R.

1 INTRODUCTION

1.1 Structure of the thesis

The central consideration of this thesis is the involvement of people in developing software for safety-critical, real-time systems. In particular, it considers how these people can accurately specify this type of system. After considering the background to this process, a new technique called animation prototyping is introduced.

In order to establish the practical applications of this technique, it is necessary to look at the type of systems at which the technique is aimed and where in the development process it is to be used. These ideas are considered in detail later in this chapter.

Chapter 2 discusses the idea of systems specifications in detail. By considering how specifications are written and what they are used for a list of useful properties for specifications and their development is suggested. The specific concerns of this thesis are:

* The pitfalls in the translation of client-written requirements into developer written specifications;

* The development of safety-critical real-time systems from these specifications.

A brief introduction to formal methods is given. This highlights why formal methods are useful in this work.

Chapter 3 introduces the idea of software prototyping. This is a diverse field and the emphasis is placed on techniques useful for the early stages of software development.

The approach to prototyping that is used here has two main strands; firstly the use of formal specification techniques and second the use of animated pictures to aid the discussion and improvement of the specifications thus written. Formal specifications are used for many reasons but above all because they provide a concrete result at the end of the prototyping effort. Animated pictures are used because they provide a more familiar medium for discussions than the mathematic formulae of the formal specification.

Chapter 4 describes what formal methods are and how they are used. It includes an overview of different techniques. The impact of the techniques on the specification process is discussed. There are still problems with an approach based solely on these techniques and these are discussed. The choice of one particular method for this work, called the Vienna Development Method or VDM, is explained. Also the idea of animation prototyping of formal specifications is introduced.

Chapter 5 gives a more detailed description of the Vienna Development Method. Particular emphasis is placed on the structure of specifications written in this style. The mathematical manipulations called proofs are one of the central features of formal methods. The purpose and results of these proofs are shown. The need for animation prototyping when using formal specifications is again shown.

Having seen the need for animation prototyping of formal specifications, chapter 6 gives a detailed description of the background issues to the development of a demonstration prototyping tool. A new specification methodology based on a structured application of the Vienna Development Method of software design (defined here as S-VDM) is shown. Also, the requirements for automated software tools to enable S-VDM specifications to control the behaviour of graphical displays ("animating the specification") are established. Finally, a diagrammatic technique for showing the structure of S-VDM specifications is also defined.

Chapter 7 shows how the implementation of the tool was achieved. This chapter shows the process of creating an animation prototype from the formal specification.

Chapter 8 gives a description of a number of animation prototypes which were built. The practical application of S-VDM for specification is shown. The animation prototyping process and the use of tools is also illustrated.

A summary of the ideas in the thesis and the scope for future work is given in chapters 9 and 10.

1.2 Real-time Embedded Systems

In order to understand the thesis presented here better it is necessary to address two questions; what sort of systems are being developed and how does that affect the development approach?

The characteristics of real-time embedded systems set them aside from many other types of systems. In these systems the correct functioning of the system is dependent on the timing of data input and output as well as the values of the data. Although computer programs are at the heart of such systems the correct performance of the system as a whole depends equally on the external non-computer

hardware in the system. In general real-time systems gather information from external sensors and then exert an influence through external devices. Examples of real-time embedded systems are chemical plant control [Williams87], windtunnel control [Williams84], astronomical data acquisition [Kelton84], fighter plane control [Kaplan85], space shuttle control [Carlow84], radar applications [Fathi84] and amusement park ride control [Nelson81].

Consider the chemical plant controller shown in Figure 1. This includes such diverse elements as the computer electronics, pressure sensors, temperature sensors, valves, pipes, pressure vessels, gas compressors and operator displays and controls. So in real-time systems there is a complex relationship between the hardware and software: with the software forming only a small part of the whole.

Let us take a closer look at some of the terms which are used to describe and classify real-time systems. The first and most obvious property of real-time systems is that they have deadlines; certain tasks performed by the systems must occur at particular times. The nature of these deadlines divides real-time systems into two groups. In the first group are systems where the failure to meet a deadline is inconvenient but not catastrophic. Systems such as this are called soft real-time systems. Examples are airline registration systems and CAE graphics systems. The second group of systems have rigid deadlines, hence the name hard real-time systems. In these systems failure to meet a deadline will lead to serious or catastrophic results. The high pressure chemical plant shown in Figure 1 is an example of this type of a hard real-time system.

Within the class of real-time systems there is a sub-class of systems with which this work is especially concerned; that of safety-critical systems. These are systems where failure of the system can endanger people's lives. Such systems include aircraft flight systems, nuclear plant controllers and high pressure chemical plant controllers. Current software engineering practice for such systems places severe restrictions on the style of software which maybe employed. These restrictions are that:-

* All programs are strictly sequential;
* There use of interrupts is forbidden;
* All Input devices, including timers, are polled.

Having outlined the systems which are of interest here, safety-critical embedded real-time systems, let us consider how software for these systems is developed.

1.3 Software Engineering.

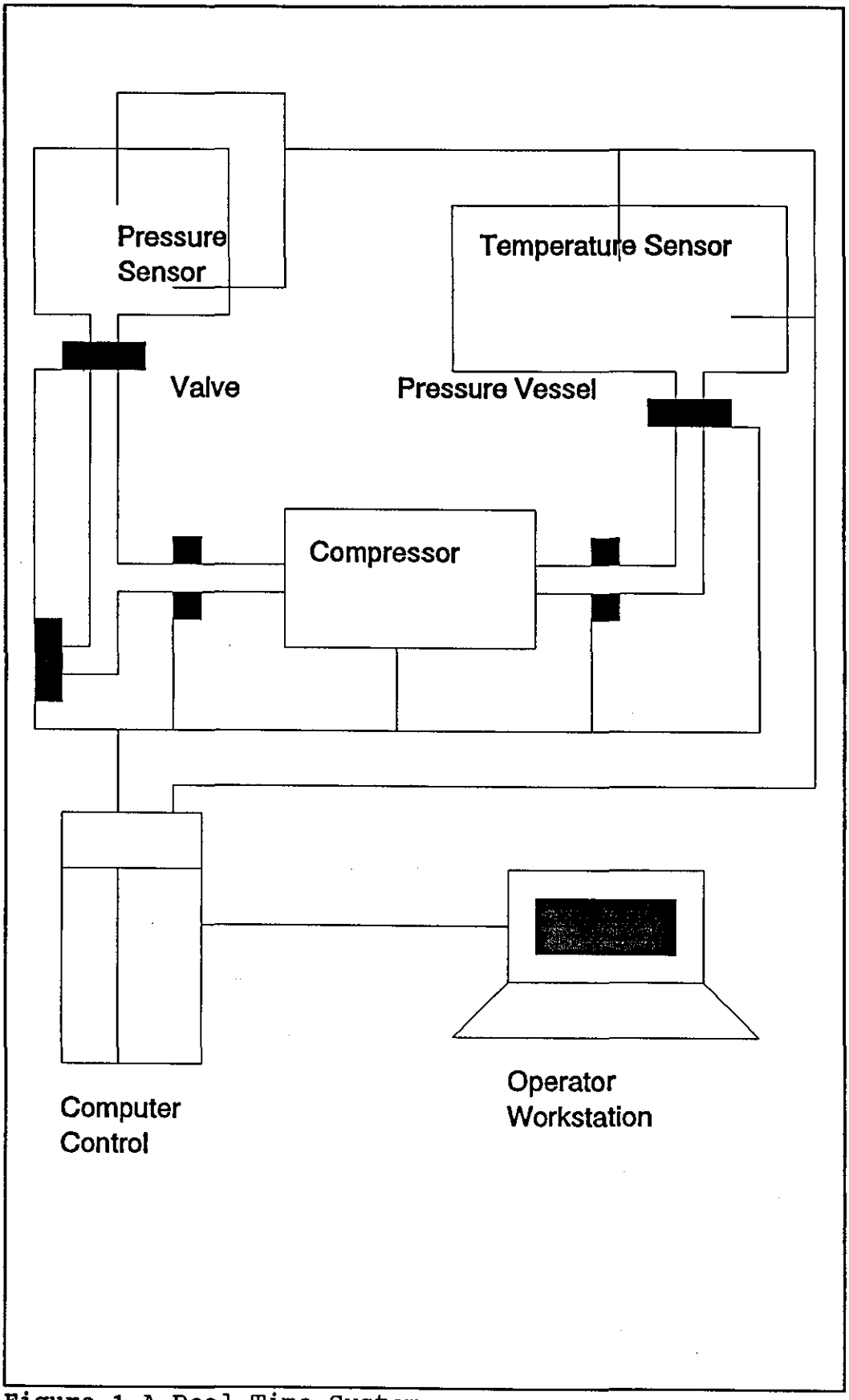


Figure 1 A Real-Time System.

Pomberger in his book [Pomberger84] defines software engineering as follows:

> "Software engineering is the practical application of scientific understanding to the economical production and use of reliable and efficient software."

The waterfall model of software development [Royce70, Boehm76] has been a popular model and it does give a useful basis for discussion. The model is shown in Figure 2. The work done at each stage is as follows:-

* Requirements specification. The problem to be tackled and goals of the project are specified. Documents are produced detailing what problems are to be addressed and constraints such as cost and timescales.

* Functional specification. Decisions are made as to what needs to be done in order to meet the project goals.

* Design. Decisions are made as to how different sub-systems will function together and what tasks each sub-system will perform.

* Coding. This is when the actual code is produced. Detailed design of sub-systems may also be a prelude to the coding of that sub-system. Sub-systems will also be tested against their functional specification.

* Integration. This is where the individual sub-systems are brought together to form the complete system. Eventually the system as a whole will be tested.

* Verification and Validation. Verification is checking the whole system to see if it fulfils is functional specification. Validation is checking the whole system to see if it matches the original requirements of the user.

* Maintenance. This is the process of modifying the system to meet the requirements of the user. This means correcting bugs not found during the development, adapting the software to cope with a changing environment and improving the performance of existing features.

There has been a great deal of criticism of this view of software development [McCracken82, Zave84, Boehm88, Luqi88, Maude91]. The major criticisms are:-

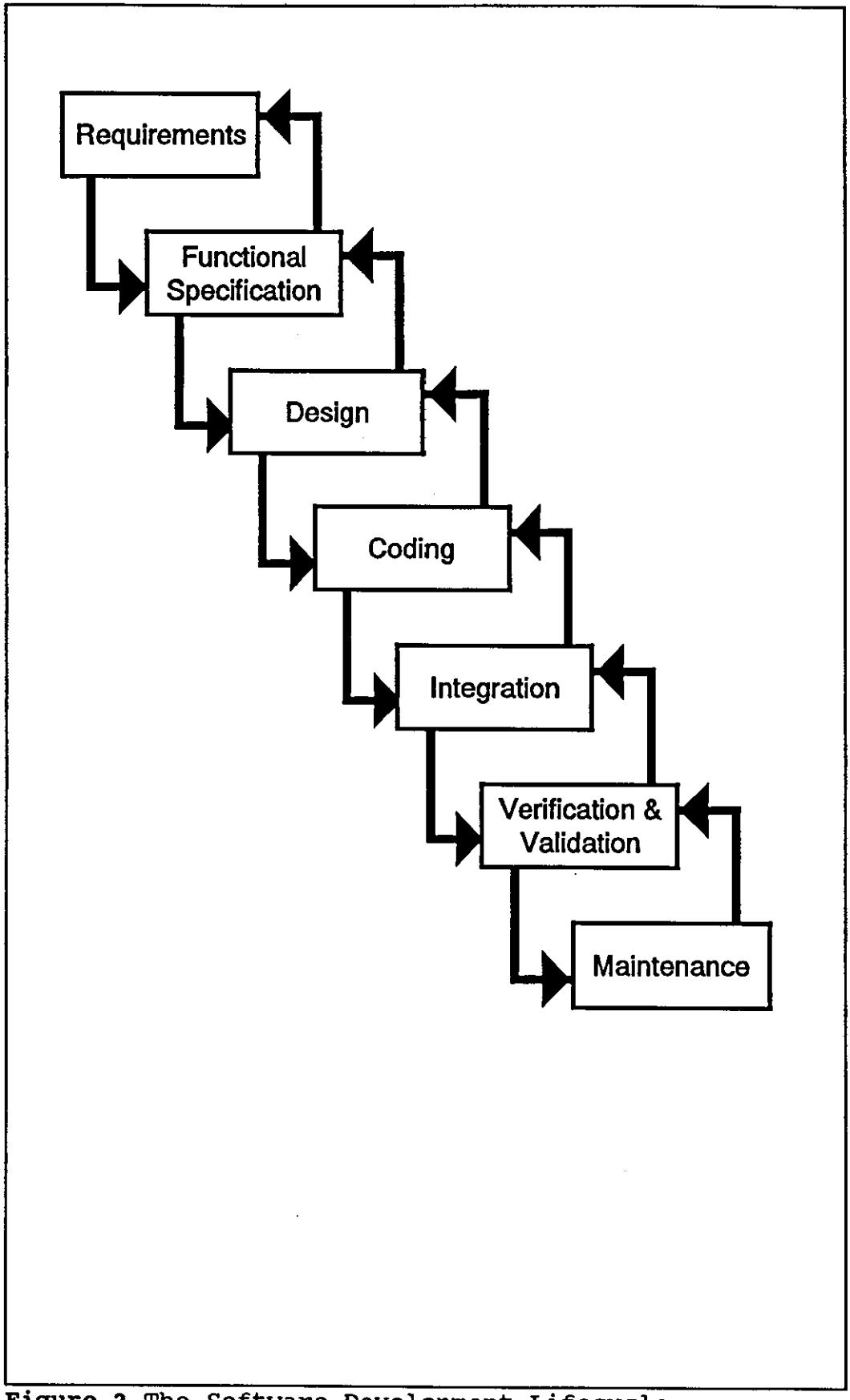


Figure 2 The Software Development Lifecycle.

* The lack of feedback between the individual stages.
* The lack of early feedback on accuracy of requirements.
* The indistinct boundaries between specifications and implementations.
* The flexible nature of requirements as project progresses.

The most serious of all these criticisms is the lack of early feedback on the accuracy of the requirements. If the requirements are not accurate then all subsequent development will achieve is to produce a system (possibly of high quality software) that performs the wrong set of tasks. A related problem is the correct interpretation of requirements to produce a specification. These ideas are considered more fully in chapter 2.

As has already been stated the communication of ideas is seen as central to this work. Between each of the above stages there is a flow of information. It is important to ask questions about this information such as:-

* Who is involved in the process?
* What information do they need?
* Who has this information?
* To who else is the information useful?
* How can the information be exchanged between parties?
* How can parties check each others view of the information?
* How can the information gathered best be passed on to subsequent stages of development?

Subsequent chapters show how animation prototyping of formal specifications of real-time systems can address these issues.

2 SPECIFICATION AND SPECIFICATION ISSUES

2.1 Introduction

In this chapter, the role of specifications within the software development lifecycle is be shown. Particular emphasis is placed on what specifications are, how they are written and what they are used for. A very important part of this process is the communication of ideas between people.

Correctly specifying the requirements of a software system is a crucial task; mistakes made here affect all subsequent stages of software development [Hughes87, DeMarco78]. Experience has shown that rectifying such errors is both costly and time consuming [DeMarco78]. As a result, many techniques have been developed to aid the specification process: De-Marco methods, CORE, SSADM, for instance [STARTS87]. These, although they are rigorous, lack mathematical formality; thus their correctness cannot be proven. Consequently, there has been a major move to introduce formal methods into the specification process [DEF STAN 00-55]. Formal methods, although they provide many benefits, do not aid communication in many situations. A new technique, animation prototyping, aims to alleviate some of these communications difficulties.

2.2 Specifications in Software Development.

The conventional approach to software development has come under much criticism [Ramamoorthy84, Zave84, Swartout82, Balzer83, McCracken82]. The main objections cited are that its divisions between the different phases of the development are too rigid. It is argued that the development process is one in which the various stages overlap and interact considerably. A very similar argument took place between the advocates of rigid engineering design methodologies and cognitive psychologists. The psychologists argued that real designers do not think in discrete stages [Snodgras88]. Therefore, they argue, design methodologies should support a wider, more flexible approach to the whole design process and the influence of decisions at all stages on the final solution should be recognised.

Balzer [Balzer83] and others recognise that in software engineering too the specification of a system is inevitably intertwined with the nature of the final implementation. Furthermore, the assumption that the specification can be fixed at an early stage in the development process is a too idealistic

[Brookes87]. The developers' perception of the customers' requirements always change. This is inevitable as the developers start with only a limited appreciation of the customers' domain. Despite this criticism, specifications are part of all software development approaches [STARTS]. In order to consider the role of specifications, three main questions need to be answered. These questions are :-

(a) What are specifications used for?
(b) Where do they come from?
(c) How are they written?

The profound effects of bad specifications on software quality cannot be ignored and consequently this is a specification-centered view of software development. As has already been stated the flow of information between groups involved in these processes is very important [Potts88].

> "(T)his brings us to the most important requirement of all: we need methods, not data flow diagrams or formal systems but real methods, methods that assist in the elicitation of expert domain knowledge and consolidation of knowledge from different sources, methods that bristle with heuristics to check the consistency, quality and appropriateness of the specification, methods that define the viewpoints from which one can inspect the specification, methods which produce specifications that can be used by clients, lawyers, requirements analysts, project managers, quality assurance auditors, hardware interface engineers, and software developers. These people are people. With so many of them involved it is time software engineers started paying more serious attention to the human factors of specification methods."

Consequently in considering the process of specification the following points are highlighted:-

* Who is involved in the process?
* What information do they need?
* Who has this information?
* To who else is the information useful?
* How can the information be exchanged between parties?
* How can parties check each others view of the information?
* How can the information gathered best be passed on to subsequent stages of development?

2.2.1 The use of specifications.

The end users of the software specification are the software designers. Their job is to decide in detail how the software will accomplish its task. They need a clear statements about function, performance,

interfaces and constraints. The specification is an expression of these goals. The overall aim of this process is to form first an architectural design and then a detailed design of the system.

The software designers are usually not the same designers as those who wrote the software specification. Consequently, as with the SOR document, the clarity and accessibility of the information in the specification document is crucial to the success of the design and subsequent phases. This requirement of specifications is emphasised when considering ways to improve the specification process.

Architectural design takes the functional and non-functional requirements and forms a structure of clearly defined software components. Each components performs a specific task which relates to the identified requirements. This is not a trivial task because of the large amount of information to be organised. In order to help them in this task there are a number of different development tools and methods. The STARTS Guide [STARTS87] contains a comprehensive review of many such software development tools.

Detailed design adds detail to the structure and components in the architectural design. This activity leads into the coding and testing of an implementation. It should also be noted that the specification will be needed by those testing the software. Their job is to ensure that the software meets all its requirements of function, performance, interfaces and its constraints. In safety-critical software, traceability between the requirements and the deliverable software is a very important. So clear, easily-accessible specifications are essential in the development of such systems.

2.2.2 The origin of specifications.

As with any engineering discipline the successful development of a software system depends on a clear understanding of the role of that system. This implies that designers must begin a project with a thorough analysis of the problem and a sound strategy for development of solutions to that problem [Prager87]. In industrial and, more specifically, military software development great emphasis is placed upon long and wordy statements of the problem. For real-time embedded systems, software specifications are generated during the overall system design phase. These specifications are defined in a **Statement of Requirements** (SOR) document, this being produced by the systems designers. In industrial and defence applications, systems designers are usually engineers, with backgrounds in fields such as mechanical, avionic, chemical and civil engineering. Software engineers are rarely employed in this role. This is one source of problems when subsequently writing specification. SOR documents are the starting point for software specification. It is essential, therefore, to know what sort of information do they contain.

According to the STARTS guide, the four main software requirements covered by SOR documents are:-

* Function - what is the software supposed to do.
* Performance - how well it should perform these tasks.
* Interfaces - how it fits in with its environment. Interfaces can be further categorised as:-
  - Physical: how people interact with the software.
    how the software interacts with external devices.
  - Software: how it interacts with other software,e.g. operating systems and databases.
* Constraints - what may and may not be done.

This information, and particularly the constraints, derive from design decisions taken even this very early stage of the development process. Real-time embedded systems are, by their very nature, only small parts of much larger systems. Consequently the overlap between specification - stating WHAT is to be done - and design - deciding HOW it is to be done - is very marked [Swartout82]. Figure 3 shows two contrasting views of a real-time system. It is apparent from this that many design constraints will have been placed on the software during the specification and design of the whole system. So the requirements for a real-time system derive from design decisions made to meet much higher level specifications. Consider the following example:-

> Managers at a large chemical manufacturing company perceive an increase in the markets demand for a new organic chemical. Their requirement for profitable production of this chemical is translated into the specification - build a production plant to produce X tonnes per year of chemical Y at price £Z per tonne. Using this specification, chemists, chemical engineers and plant engineers decide on what method of production to employ. Their requirement for physical devices to implement this process becomes a number of new specifications such as - a storage vessel to hold W tonnes of a chemical, piping to withstand V KPa, heaters to produce U watts, a temperature controller, a valve sequencer and a pressure controller. Some of these specify the physical plant while other devices seem to have functions which may require microprocessor control. One such device, a temperature controller, is to be implemented using a microprocessor. Finally a specification for real-time software is generated.

This example serves to illustrate the very close link between real-time software and the larger system of which it is part. The specification for real-time software is the result of a number of higher-level design decisions. These decisions are taken by people who are not necessarily software engineers. The software to be developed must meet the requirements of these people and thus software engineers

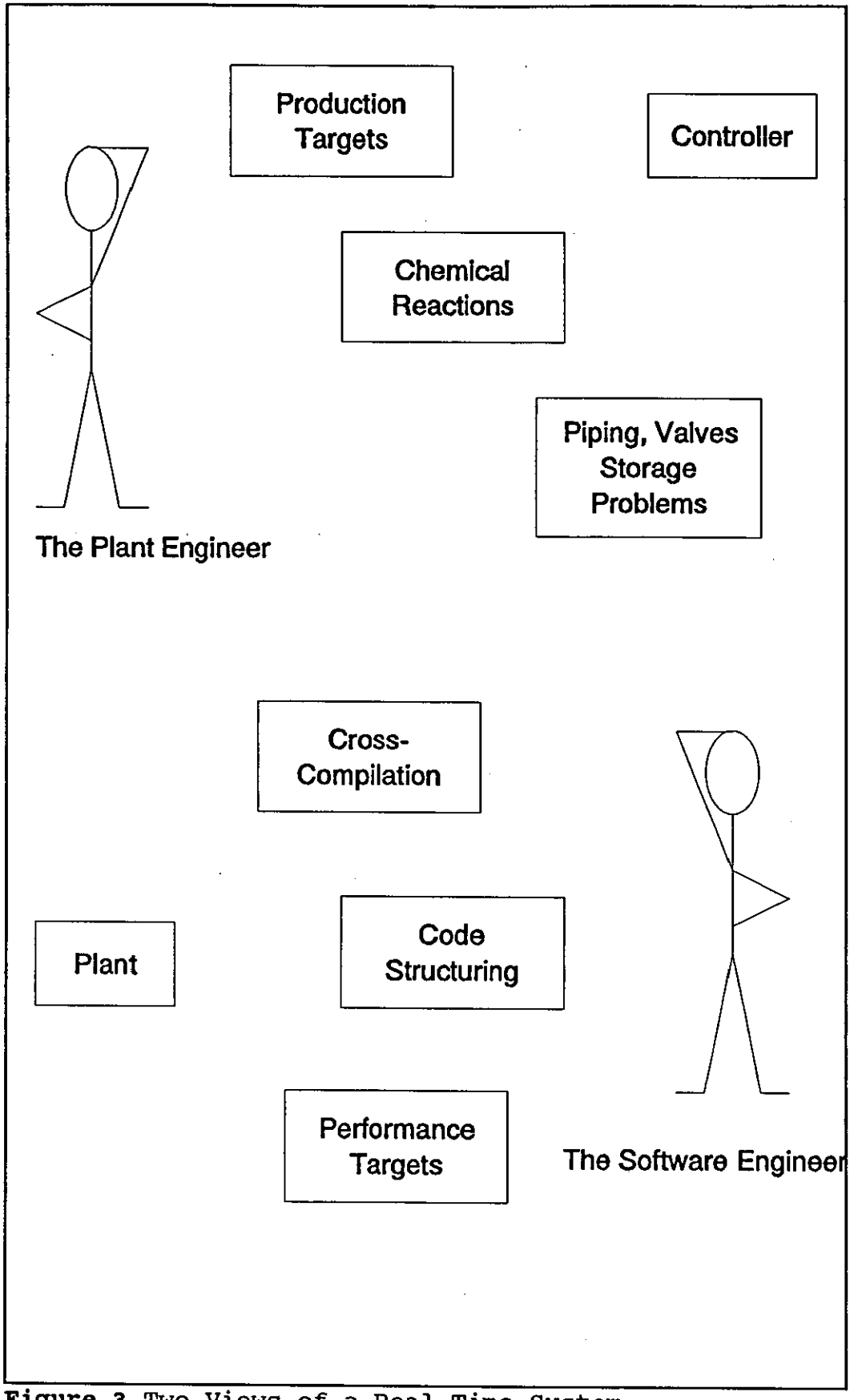


Figure 3 Two Views of a Real-Time System.

must be able to communicate their thoughts effectively to non-software engineers.

So in summary, before a specification is written, the system designers:-

* Analyze the problem thoroughly from their own point of view;
* Form a strategy for developing a system to solve that problem;
* Write a statement of requirements document stating the problem and the required form of solution.

This statement of requirements document is used by the software designers as the basis for their software specification.

2.2.3 The writing of specifications.

There is a great deal of information passed from system designers and engineers to software designers in the SOR document. Software specification is the process by which software designers assimilate and organize this information for use by software engineers. There are a number of problems which make this a difficult task.

The first hurdle is the use of specialist terminology. Systems engineers have a good understanding of the system domain: function, structure and behaviour. SOR documents are written from their point of view, that is, the external environment of the software. Normally they are expressed using the technical language and terms of the application environment. The software designer, by contrast, is unlikely to have such detailed knowledge. Thus he is faced with the problem of interpreting the requirements of the specification; then translating these into a form which can be used during the software design phase. This task can be difficult enough when the requirements documents are clear, precise and correct. Unfortunately, as they are frequently ambivalent, ambiguous, incomplete, and sometimes in error [Meyer85], it may be an extremely onerous task. This is the second hurdle in the development of good quality specifications.

Technical language is not the only problem with SOR documents. A clear understanding of the requirements is obstructed by the text of the SOR being:-

(a) Ambivalent. The SOR is written so that it can mean one thing or another or possibly both.
(b) Ambiguous. The SOR fails to make its point clearly enough for the reader to understand.
(c) Incomplete. A vital, or useful, piece of information has been omitted.
(d) Inconsistent. The SOR includes contradictory information.

(e) Complex. The SOR contains all necessary information, but in such a way as to make comprehension difficult.

(f) Wrong. The SOR contains information that is not true. Such errors derive from many different sources.

Such problems can only be resolved satisfactorily by consultation with the system designers. This is also not a trivial exercise. Discussions between the system designers and software designers can also highlight further problems with the SOR and its interpretation:-

(h) What the SOR describes is not always what was intended by the system designers or is actually needed by the user.

(i) The software designers have incorrectly interpreted what the SOR says.

From this it can be seen that developing a good quality specification is a highly interactive process. Figure 4 shows a simple model of the interpretation of the SOR to build a specification. This illustrates the importance of discussions between the parties. Through these discussions and the increased understanding that develops, both the specification and the SOR are improved.

2.3 Ways to Improve the Specification Process.

The central theme of the specification process is the communication of ideas and information between different groups of people involved in the software development process. Specifications are essentially a bridge between the systems designers requirements and the software designers efforts to build software to meet them. Any improvements to this process must thus aim to smooth the passage from SOR to specification and thence to software design and testing.

The transition from SOR to specification is a highly iterative process. Improvements to this transition process include:-

(a) Careful writing of SOR documents.

(b) Using good structure so that requirements are easily accessible to both systems designers and software specifiers.

(c) Explanation or avoidance of highly specialised technical language.

(d) Review of specifications with respect to the SOR. The software specifiers need to express their understanding of the requirements clearly to the system designers.

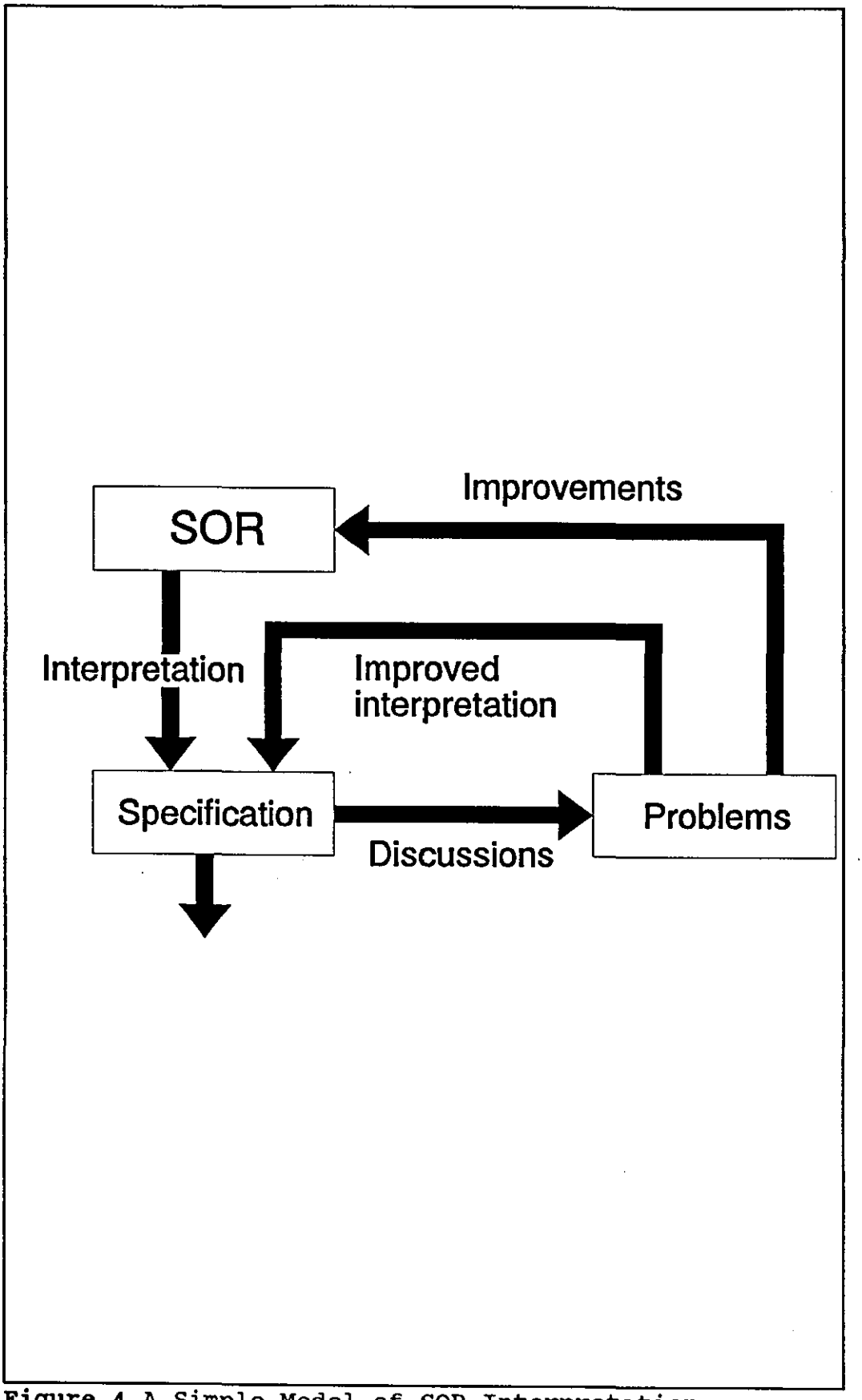


Figure 4 A Simple Model of SOR Interpretation.

Producing specifications which can analysed systematically is another way to aid the specification process [Ramamoorthy84]. When the specification is being produced the specifiers need to answer certain questions. They also need to convince the system designers that such questions have been answered. Some of the questions which might be posed by the specifiers are listed below.

What does the specification say?

Is it consistent within itself?
Is it consistent with the specifiers' view of the system?
Is it consistent with the system designers' view of the system?

Is it complete within itself?
Does it contain all the information needed?

Is it correct within itself?
Is it syntactically correct?
Is it semantically correct?
Is it feasible?

Is it correct?
Does it describe the system as the specifiers understand it?
Does it describe the system as the system designers understand it?
Will the specified system perform the desired tasks?
Will the specified system meet safety requirements?

These are important points for the specifier, but the management of this process is equally important. Any approach to specification ideally should support :-

(a) The coordination of teamworking;
(b) The documentation of benchmarks for acceptable functionality and performance;
(c) The ability to express design decisions;

These points aid the writing of a specification which accurately expresses the system designers requirements. However, the software designer as end users of the specification need considering. In order to aid the production of software in line with the requirements, a specification must state the requirements :-

(a) Unambiguously. All requirements must be carefully written so as to avoid different interpretations.

(b) Completely. All significant information must be provided. Terms must be clearly defined.

(c) Verifiably. Where possible measurable quantities or properties required should be provided.

(d) Consistently. Statements about requirements must not conflict.

(e) Traceably. The origins of each requirement should be clear.

To overcome some of these problems, and to improve software quality and productivity, the use of mathematically based specification languages, or formal methods, has been proposed [Gibbons87, Cohen82]. A second technique, software prototyping, has been proposed in order to address the problems of communication and requirements analysis [Budde84]. The next two sections give a brief introduction to these techniques and suggest how the two can be combined.

2.4 The Role of Formal Methods in Specification.

Formal methods are based on mathematical formal systems. They use a mathematically based formal notation and reasoning system to describe and analyze the structure, function and behaviour of software systems. Carefully applied to the development of software, they form the basis for a rigorous engineering approach [ALVEY84].

The role of formal notations in software engineering varies considerably. Some methods are intended to cover a large portion of the software life cycle. VDM [Jones90], for instance, embraces the cycle from specification through to coding. Others, such as FOREST [Goldsack88], address themselves only to the problem of stating system specifications. Here the concern is mainly with the specification aspects of formal methods.

There are a number of claimed benefits for the use of formal specification techniques [Hall90]. First, they introduce precision, rigour and clarity of thought into the process. As a result, the specification document is likely to be correct, consistent and complete. Second, the document itself can be used as a firm basis for interaction between the software designers and the system specifiers. Third, this approach also raises the visibility of the project as documentation is produced from the beginning of work.

The application of this approach raises questions as to who should produce the formal specifications. Few systems designers have knowledge or experience of discrete mathematics and formal reasoning,

the basis of formal specification techniques. Consequently, it becomes necessary to employ specialists to convert the informally expressed requirements of the system designers into formal software specifications. When they complete their work the resulting formal specifications are presented to the system designers for approval and agreement. But the notation used is fully comprehensible only to the experts who produced the formal specifications. The systems designers must decide whether their requirements are being correctly specified. This, at the present time, is a major problem for the designers of real-time embedded systems.

A final - crucial - point concerns the use of formal methods for proving the correctness of specifications. Proponents of this approach stress the confidence obtained by using mathematical techniques in place of conventional procedures. But rarely are the complexities of proof discharges highlighted. To illustrate this point, Appendix D contains the discharge of the implementability proof for the very simple logic function specified there. Its intricacy raises a serious question. How confident can we be in our ability to detect errors in such mathematical workings?

The need to make formal specifications comprehensible to non-specialists is great. The recognition that system designers need to assess the accuracy of formal specifications is the main motivation behind the approach to specification called animation prototyping.

2.5 The Role of Animation Prototyping in Specification.

The approach used in this project is called animation prototyping. The essential objective of animation prototyping is to illustrate the key properties of specifications to non-computer specialists by using computer-animated pictures [Cooling89]. It provides a demonstration of executable specifications ('animates the specification') in terms of the SYSTEM domain. Animation prototyping may be used to express both the SOR objectives and those of the formal specification document. In this way the system and software designers are more easily able to evaluate the interaction of the software with its environment. This interaction leads to a greater understanding of the system objectives and operation; in turn this should result in software which more accurately meets the needs of the client.

Commercial software projects place severe restrictions on development time and cost. For animation prototyping to be acceptable in this sort of environment, it must fulfil three main objectives:-

(a) Model production must be done quickly - typically no more than a few weeks.

(b) Clients should need only a minimal knowledge of technical jargon to understand the model behaviour.

(c) Both client and developer must be able to interact with the model and thus increase their mutual understanding of the problem.

In the commercial development of software, hand-built prototypes are prohibitively expensive. It is therefore necessary to consider what techniques and tools can be used to help in the model building. On this basis, the approach suggested here is to use an executable program, derived automatically from the formal specification, to animate pictures on the computer screen. In this way, animation prototyping aims to help in the discussion and understanding of the problem and the ideas expressed in the formal specification. Animation prototyping and other forms of software prototyping are discussed in more detail in chapter 3.

3 PROTOTYPING AND ANIMATION OF SPECIFICATIONS

3.1 Software Prototyping.

3.1.1 Prototypes in engineering.

Looking at other engineering disciplines can give an interesting perspective on what prototyping means. In the aerospace industry, full scale working prototypes of aircraft are commonplace. Every year automobile industrial shows are filled with the latest prototype designs. Much electronic design these days is accomplished through computer aided design packages. These offer not only tools for laying out designs but also functional simulators. Thus an engineer can see some of the important features of a design before the actual production work commences. It is worth mentioning that these are disciplines where the investment in production equipment is considerable. Thus it is necessary to know (or have a high degree of confidence) that a design works before production commences. However the production costs of software are escalating to levels where such confidence building is becoming essential.

In software engineering the use of prototypes has also been advocated. Software prototypes have been variously described:

> "Prototypes present the user with a relatively realistic view of the system as it will appear." [Mason83]
>
> "A prototype is an executable model or pilot version of the intended system." [Luqi88a]
>
> "A system that can solve parts of a problem and is used to show potential value to management and prospective sponsors." [Jordan89]

These quotes, particularly the last one, could be made by engineers from any field. It can be seen that prototypes have many possible uses:

(a) To test the feasibility of various different design approaches.

(b) To test particular technical aspects of a design (eg Space shuttle Enterprise which has no engines but mimics the glide characteristics of the full working shuttles)

(c) To attract funding for further development work (eg Experimental Aircraft Project).

(d) To test user or customer reaction to a particular design (eg New car designs).

3.1.2 Prototyping and the Software Life Cycle.

Software engineers should adopt prototyping essentially because they suffer from the same problems which other engineers use prototyping to solve. The engineer's job is to efficiently build a system to solve a customer's needs. In real-time embedded systems, the "customers" are the systems designers. It is they who produce the SOR document and it is their requirements which must be satisfied. There are a number of key aspects to this problem:

(a) Correctly identifying the problem to be solved.
(b) Generating possible approaches to solving this problem.
(c) Using professional judgement to select the "best" approach.
(d) Overseeing and helping with the resolution of detailed technical problems encountered in the realization of a solution using the chosen approach.
(e) Ensuring the efficient production of a quality product for delivery to the customer.

Effective methods of code design and production enable software engineers to fulfil the last two objectives. Better methods of dealing with the first three issues are now needed.

As discussed in the previous chapter, there have been a number of different proposals as to what should replace the conventional life cycle model. The one on which the work described here is based is software prototyping.

3.1.3 Prototyping and the Specification Problem.

The largest and most important task facing software designers is to establish the customers' requirements for the system. It has been shown by De Marco [DeMarco78] that problems are caused by errors in the writing and interpretation of the statement of requirements document. Furthermore, these errors are very expensive to correct in terms of time and money [DeMarco78]. Brookes [Brookes87] argues that a system cannot be correctly specified without the customer being able to test a working version of the product. Proponents of rapid prototyping support these views and aim to provide a cost effective means of producing a demonstratable version of the design concept [Maude91].

As seen in the previous chapter, specifying a real-time system is made more difficult by

communications difficulties between system designers and software specifiers. The interpretation of the system requirement is requires a great deal of discussion between the people involved. With the differing perspectives involved it is one in which misunderstandings easily occur. There is a need for all parties to be able to explore a commonly agreed definition of the problem. This requires a method of defining the problem which is easily understood by both parties. This is one of the major reasons for adopting the use of animation prototyping.

3.1.4 Different Types of Software Prototyping.

Software prototyping is a relatively new development in software engineering. There are many different types of software prototyping. Various classifications have been proposed for the different types of software prototyping. Watkins [Watkins88] proposes a useful definition as shown in Figure 5. Although it depends a little too much on the reference to a conventional software life cycle model, Ratcliffe's [Ratcliff88] definition places more emphasis on WHEN prototyping occurs than WHAT it is intended to achieve as shown in Figure 6. Schneider [Schneider87] provides a definition based on stages and levels of prototyping. He states the distinction between stages and levels is:

"(T)he stages of prototyping deal with differences in kind, while the levels of prototyping deal with differences in degree."

The relationship between levels and stages of prototyping as envisaged by Schneider are shown in Figure 7.

The common factor in all software prototyping is to allow more customer involvement in software development. This involvement is aimed at producing software which meets the customer's requirements more closely than conventional methods allow. The vehicle for this involvement is usually a demonstration of part of the systems proposed functionality or external appearance. Here the major concern is with using prototyping to extract customer requirements. The techniques of interest is a type of early prototyping called animation prototyping. The work of Budde et al [Budde84] and Tanik and Yeh [Tanik89] gives a broader view of other types of software prototyping.

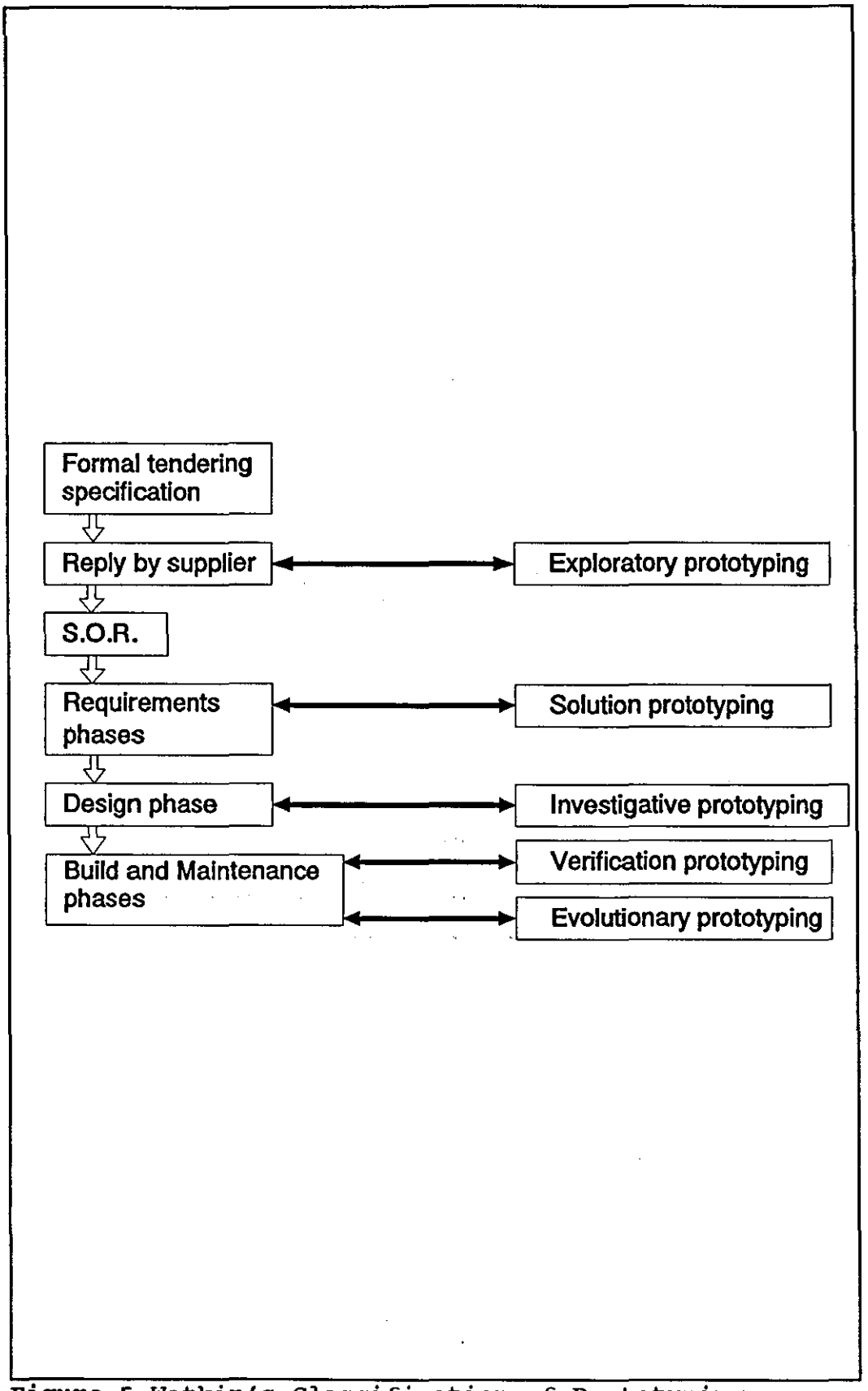


Figure 5 Watkin's Classification of Prototyping.

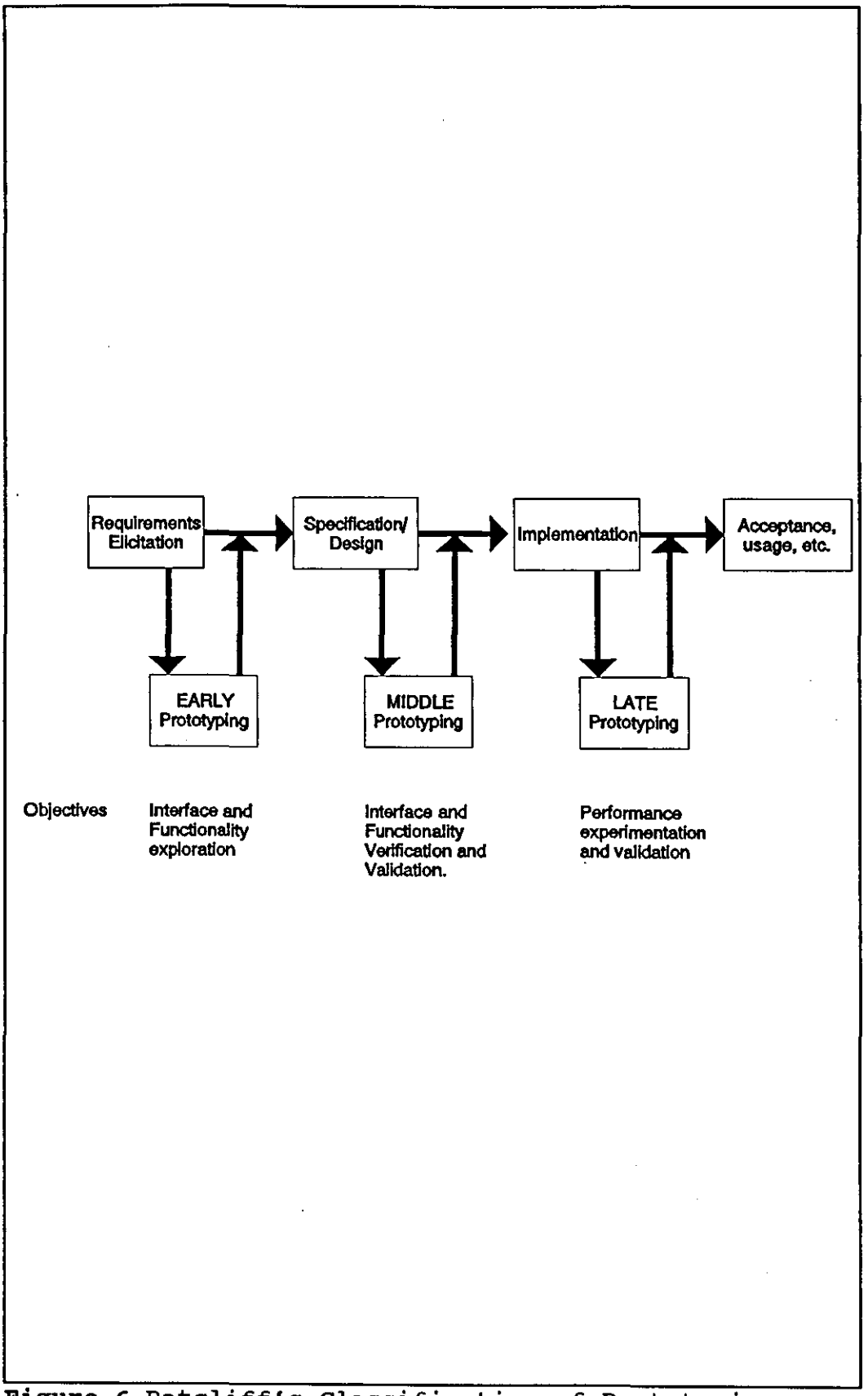


Figure 6 Ratcliff's Classification of Prototyping.

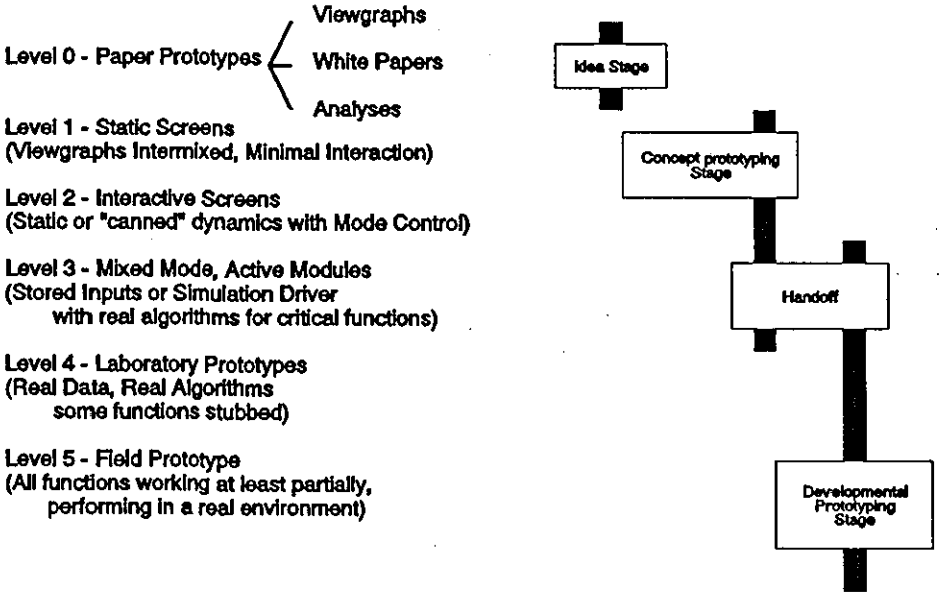


Figure 7 Schneider's Classification of Prototyping.

3.1.5 Constraints on software prototyping.

Prototyping to explore customer requirements occurs at a very early stage in the design process. Thus any method used should be economical, fast and adaptable, for the following reasons.

(a) Economical: It is possible that the outcome of requirements analysis may be that the system is not economically viable in its present state. Whether the project is abandoned at this stage or pursued in a modified form, work upto this stage must be scrapped. Therefore it is essential that prototyping should not absorb great amounts of resources.

(b) Fast: At this stage answer are needed quickly, especially prior to and during the tendering phase of the project.

(c) Adaptable: As the development of the prototype is an ongoing process, any methodology must provide for easy modification of the model.

If these three goals are not achieved three problems may be encountered:

(a) Slow response to customer's enquiries. This may undermine confidence in the developers.

(b) Lack of flexibility. This may lead to a waste of effort in trying to reuse parts of an inadequate model.

(c) The focussing of excessive efforts on the development of the model. Prototyping should never be seen as a replacement for design techniques; rather it aims to increase the effectiveness of such techniques.

Methods designed to support the quick building and evaluation of prototypes are called rapid prototyping.

3.2 Rapid Prototyping.

The concept of rapid prototyping comes from the field of interactive information handling systems. Work here showed that to be effective prototyping must be supported by extensive tool sets [Musa85, Gomma81, Dearnley83, Alavi84].

If software prototyping is to be widely adopted it must demonstrate clear commercial advantages. Software production is a competitive business. Thus techniques which are expensive in terms of time and money won't be accepted unless it can be demonstrated that the rewards for such investments are commensurately large. Thus there are two main reasons for using "rapid" prototyping.

(a) Reduction of the time spent building prototypes. This makes rapid prototyping a commercial proposition.

(b) Improving the ability of the software produced to meet client requirements. This is very important in many real-time applications, particularly safety-critical applications.

In rapid prototyping models are built quickly then demonstrated to the client for evaluation. The results of this evaluation are then used to modify the model. This process is repeated until the customer is satisfied about the properties demonstrated by the model. A typical scheme for rapid prototyping is given in Figure 8. In rapid prototyping a question which needs careful consideration is how customers are to be involved. In the words of Potts, in an piece of work highly critical of software engineering methods [Potts88].

> " It is no use 'executing' a specification only to watch it wobble bewilderingly; the behaviour of the specification must be explained in terms of application-specific constructs with which the client is familiar."

The approach adopted in this research project is to use computer-animated demonstrations of the system's interaction with its external environment. This is called animation prototyping.

3.3 Animation Prototyping.

3.3.1 An introduction to animation prototyping.

There are two central themes in animation prototyping. The first is a model which is used to demonstrate important properties of the proposed software. The second is the provision of tools and methodologies to facilitate the quick construction and manipulation of these models. Animation prototyping is aimed at the specification of real-time embedded systems.

Building a model is considered important because:-

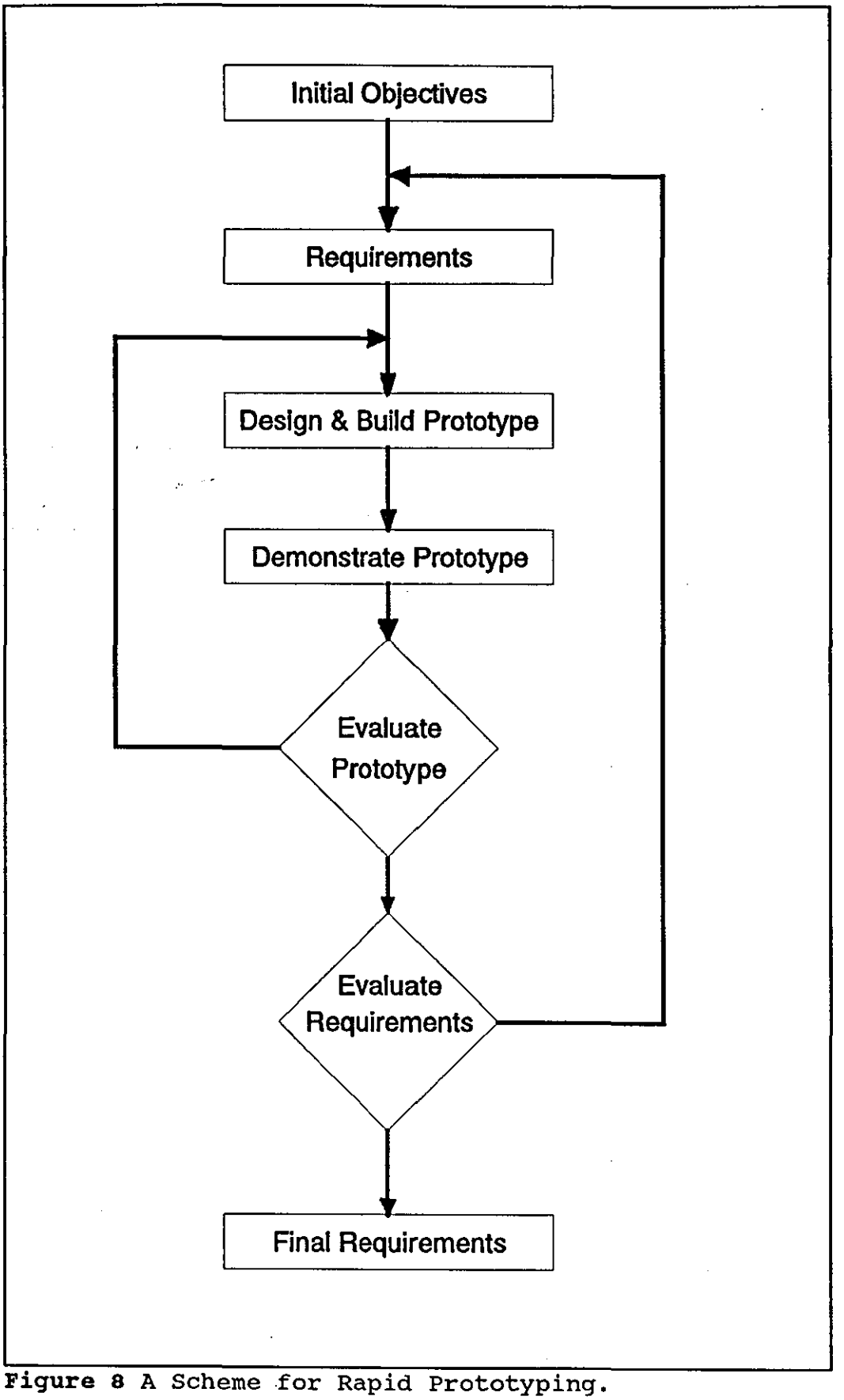


Figure 8 A Scheme for Rapid Prototyping.

(a) It acts as a vehicle for communication between customer and developer. The specification must be assessed to see if it describes the customers' requirements. Real-time systems are used in areas where the customer has very specialised knowledge about his own field but very little knowledge of software engineering. Model building makes the specification comprehensible to non-software specialists.

(b) It allows both parties to increase their understanding of the problem. In particular, by demonstrating the dynamic properties of the system, key elements can be identified. This is of special importance when dealing with real-time systems where dynamic responses are complex.

(c) The explicit modelling of the system helps to avoid misunderstandings about the importance of response requirements. In some cases these requirements are fixed by plant or safety considerations and are beyond the control of the software developer. In other cases requirements can be changed to eliminate unnecessary, costly design requirements. Such problems arise when customers do not appreciate the technical difficulty of achieving a response which is not critical to the system's safe functioning.

In data processing applications, where systems are oriented around the user interface, generating feedback from customers about the "feel" of a system is relatively easy. There are certain features which are common to most data processing systems and they have all spawned specialised prototyping tools. Hartson's paper [Hartson91] gives a good "state-of-the-art" report on rapid prototyping in this field. In real-time systems these features are of lesser importance. Of far greater importance is the question of the system's interaction with its external environment.

In order to obtain customer feedback it is necessary to demonstrate the model properties. This must not require customers to have a detailed knowledge of software engineering techniques. When rapid prototyping real-time systems, a mimicking of the system's actions is often the best way to demonstrate to the customer what the proposed software does. However, unlike data processing problems, there are few recurrent themes. Tools for real-time animation prototyping must support the modelling of systems with very diverse behaviour.

3.3.2 Uses of animation prototyping.

Animation prototyping is seen mainly as a vehicle for communication between customers and developers. These discussions are aimed at a number of different areas where communication

difficulties exist.

(a) Tendering for contracts. The production of animated prototypes to demonstrate possible solutions to customers has advantages:

* Demonstrating understanding of the customer requirements as currently expressed.
* Demonstrating in a non-technical manner the results of using different software techniques.
* Demonstrating particular products or expertise that the developers possess.
* As a side effect, producing a useful introduction to the project for new team members.

(b) Requirements Analysis. To successfully analyse customer requirements, developers need a means to express their ideas about a system. Animation prototyping can be used to provide animated models of basic concepts.

(c) Validation of Formal Specifications. As recognised by Potts [Potts88] one of the greatest problems facing users of formal methods is communication with the customer. More importantly large formal specifications are very difficult to interpret by any single person. This problem is especially important in organisations where the people who generate the informal system concepts do not understand formal notations.

Consider the example in Figure 9. Assessing the formal statement to see if it accurately expresses the original idea is crucial. The use of animated prototypes translated from formal problem statements can overcome this problem. The use of animation prototypes in this role is the main focus of this thesis and is discussed in greater detail later in this chapter.

Animation prototyping is a technique which can enhance communication of the properties of dynamic systems. It places communication with the customer at the heart of its approach to software development.

3.3.3 Rapid prototyping techniques and animation prototyping.

Building models of system behaviour quickly is central to animation prototyping. With large, complex systems this is not an easy task. Current researchers in rapid prototyping have made use of new developments in software engineering technology to help prototype production. The most widely used technologies are:-

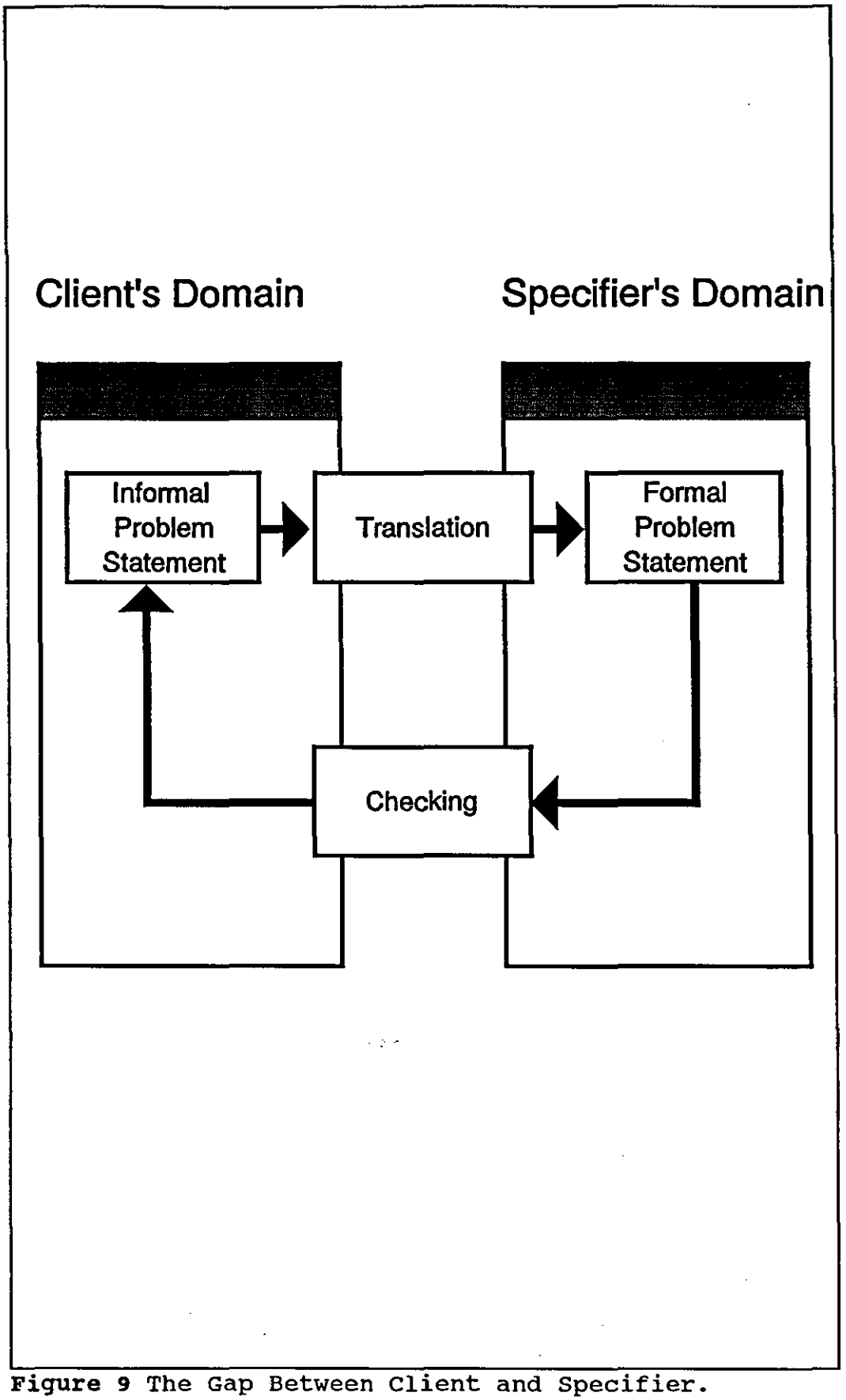


Figure 9 The Gap Between Client and Specifier.

* Object-oriented programming and design.
* Executable specifications.
* Knowledge elicitation and representation.

The following sections deal with each of these techniques in turn. It also details work on rapid prototyping carried out using each technique.

(a) Object-oriented programming and design.

This approach to software structuring proposed by Grady Booch and others has recently become very popular. Object-oriented design is best described by Booch [Booch86]:

> " Simply stated, object-oriented development is an approach to software design in which the decomposition of a system is based upon the concept of an object. An object is an entity whose behaviour is characterised by the actions that it suffers and that it requires of other objects."

Object-oriented design concentrates on the objects which exist in our model of reality. When decomposing a problem in this fashion concurrency is expressed as it naturally occurs. For real-time systems this is very advantageous. In animation prototyping the key issues are rapid model building and flexibility. Object-orientation addresses these problems by compartmentalising a design into objects. According to Booch object visibility within a system can be restricted. This means that changes to the behaviour of an object should have only localised effects on other objects. This property leads to a greater freedom to modify models which is an important aspect of rapid prototyping.

Many authors on OOD claim that this approach inherently gives rise to reusable components. They propose that building a system should consist of recombining existing objects and writing a few new ones. However, controlling the huge libraries of objects required to make this possible is beyond current technology. Object libraries on a large scale pose special problems, outlined by Jones [Jones88], which have yet to be tackled.

There have been a number of suggested applications of this technology to rapid prototyping:-

* The simplest approach to rapid prototyping is to use an object-oriented programming language, such as Smalltalk, to produced hand-built, disposable prototypes [Sandberg89].

* A more sophisticated approach is to have an toolbox full of objects [Kreutzer90]. These objects are designed to give the programmers much greater support for building prototypes. However such prototypes are still hand-built and disposable.

* Of greater interest is the development of object-oriented requirements specification languages for real-time systems [Diaz-Gonzalez88, Bruno86]. These languages have in turn spawned tools to support the building of animated prototypes [Diaz-Gonzalez89, Baldassari88]. The ENVISAGER system described by Diaz-Gonzalez is interesting in that animations use pictures of the system which would be familiar to the user. The work of Baldassari uses animated Petri nets. This work has much in common with the executable specifications described in (b)(ii) below.

(b) Executable specifications.

Much modern software design is achieved through the use of structured design techniques. Many of these techniques, such as YSM [Ward85, Hatley88, Yourdon89], JSD [Cameron86] and others, have diagramming formalisms to express system and software structures. Other techniques, such as the formal methods (see chapter 4) and PSDL [Luqi90], are based on specially designed specification languages. A number of different authors have proposed systems which allow these specifications to be executed. It is important to note that executing a specification of a system does not necessarily produce actions visually comparable to that of that system. Execution can mean a number of different things:

(i) Question answering: such as "what result do I get if this input is given?" and in more complex systems "Is there a path which connects this input port to this output port?" are permitted [Pacini87]. Such systems are usually based upon the translation of a specification language into a function language such as LISP, ML or PROLOG. Alternatively the specification could be written using an expert shell [Jordan89, Noren88]. The interaction with the prototype is text-based. This work is closely related to the execution and animation of formal specifications which will be discussed in chapter 4 where formal specifications are dealt with in greater detail.

(ii) Animated structure diagrams. In these systems tokens are displayed to represent data moving around a data flow diagram. Also some systems incorporate state transition diagrams where the current state is highlighted. Different specification systems which have been investigated for animation are given below:-

* Data Flow Diagrams, Control Flow Diagrams and State Transition Diagrams as described by Ward-Mellor [Blumofe88, Coomber90]
* Jackson System Development [Adhami88]
* STATEMATE [Harel90, Smith88]
* CORE [Kramer88]

The main drawback of these systems is that they use diagrams familiar to software engineers as the basis for animation. This makes them a good communications vehicle for software engineers exploring and testing a specification. However, these diagrams may not be meaningful to non-software engineers and it is with them that serious communications problems exist.

(iii) Prototype code execution. In this type of system a specification is written in a specially designed language. The specification is then translated into an conventional language. The resulting code is then executed and used in a conventional manner [Luqi90].

The work done by Luqi and Berzins provides is the most advanced work in this field. They have developed an "object-oriented prototyping language" [Berzins88] called PSDL in which specifications are written. The aim of the language and its supporting tool set is to allow rapid prototyping of real-time software. This approach supports the exploration of timing relationships within software for large real-time systems. The authors claim that PSDL provides a powerful abstraction technique for describing real-time systems [Luqi88a].(b) The specification can be executed by translating it into software primitives written in a conventional programming language (the authors use ADA) [Luqi88c]. To aid in this translation, the system provides for automatic retrieval and combination of software components [Luqi88d, Luqi90].

(c) Parallel work in knowledge-based systems.

Requirements analysis and specification deals with the expression of a problem. The developer is trying to ascertain what the customer knows about the problem. Research in knowledge-based systems is also concerned with the expression of knowledge about a problem and how to solve it. It has been suggested that rapid prototypers, particularly those involved in requirements analysis, should study and learn from work which has been done in this field according to Zualkernan [Zualkernan88]. Of special interest are the issues of knowledge representation and knowledge elicitation.

Knowledge is not an easy concept to understand. Many good examples of the difficulties faced in

expressing knowledge are given by Dreyfus and Dreyfus [Dreyfus86]. They illustrate their point by asking what one "knows" about how to ride a bike. The ability to ride a bike is commonplace, but expressing the knowledge of how to do it is exceptionally difficult. The most important lesson to be learned from this is for software engineers to recognise the human side of their problems.

Zualkernan proposes that knowledge acquisition techniques developed by researchers in the field of Artificial Intelligence could be useful for extracting requirements from the customer. He correctly points out that methods for knowledge acquisition such as protocol analysis [Ericcson84, Johnson87] are far more sophisticated than those currently used by software engineers.

A rapid prototyping systems which has arisen out of work done in the field of knowledge representation is FRORL. The FRORL system proposed by Tsai, Aoyama and Chang [Tsai88] uses the concept of frames and rules to express requirements. The authors identify the aims of their language as follows:

(i) Allow requirements engineers to specify objects in the corresponding application domain, also the possible changes, constraints and assumptions in the world.

(ii) Allow requirement engineers to describe their concepts about the world.

(iii) Provide a mechanism for data abstraction and the capability for the stepwise refinement process.

(iv) Permit completeness and consistency checking of the requirements.

The system appears to be a very useful tool for examining the correctness of a set of requirements.

St-Denis [St-Denis90] describes a similar piece of work based on a knowledge-driven systems. The system again has its own specification language CML. More interestingly the author has carefully considered the presentation aspects of the animation. The example of a lift system given shows an animated specification that would be meaningful to non-software specialists.

The concept of frames and rules for the expression of requirements is a sound one, stemming as it does from a great deal of research carried out in the field of knowledge-based systems. With the appropriate tool support it should be possible to explore the properties of a set of requirements very thoroughly from a number of different perspectives. Some authors [Kramer88, Loucopoulos89] have suggested schemes where the tools provide "intelligent" help facilities. They claim that such systems can provide active guidance to help in the analysis of requirements.

3.3.4 Concluding remarks on rapid prototyping.

Of all the approaches to rapid prototyping explored the executable specifications techniques provide the greatest hope for the near future for four reasons:

(i) They support structured design techniques. This will provide managers with a better degree of control over prototyping efforts than they have previously had.

(ii) Prototypes derived automatically from specifications will be much cheaper to produce then specially built prototypes.

(iii) The use of structured design techniques will make incorporating the results of the prototyping process into subsequent stages of software development easier.

(iv) The use of a single unified approach to software development within an organisation should be of great benefit in terms of tooling costs and personnel training costs.

The above comments also apply if the specifications executed are based on formal notations. The work here is based on the use of formal specifications. The basic approach and other related work is discussed in more detail later in this chapter.

3.4 Key issues in animation prototyping.

3.4.1 Experience of animation prototyping.

In the light of the experimental work described in Appendix A, a much clearer view of the processes involved in animation prototyping can be seen. In particular, any approach to animation prototyping must address three inter-related issues. These issues are shown in Figure 10. In formulating a practical approach to animation prototyping a number of important questions relating to each issue are posed. They are outlined in detail below.

3.4.2 The model building.

At the heart of animation prototyping is the idea of building a model of system behaviour. The specifiers need efficient techniques to build potentially large prototypes. Model building raises a

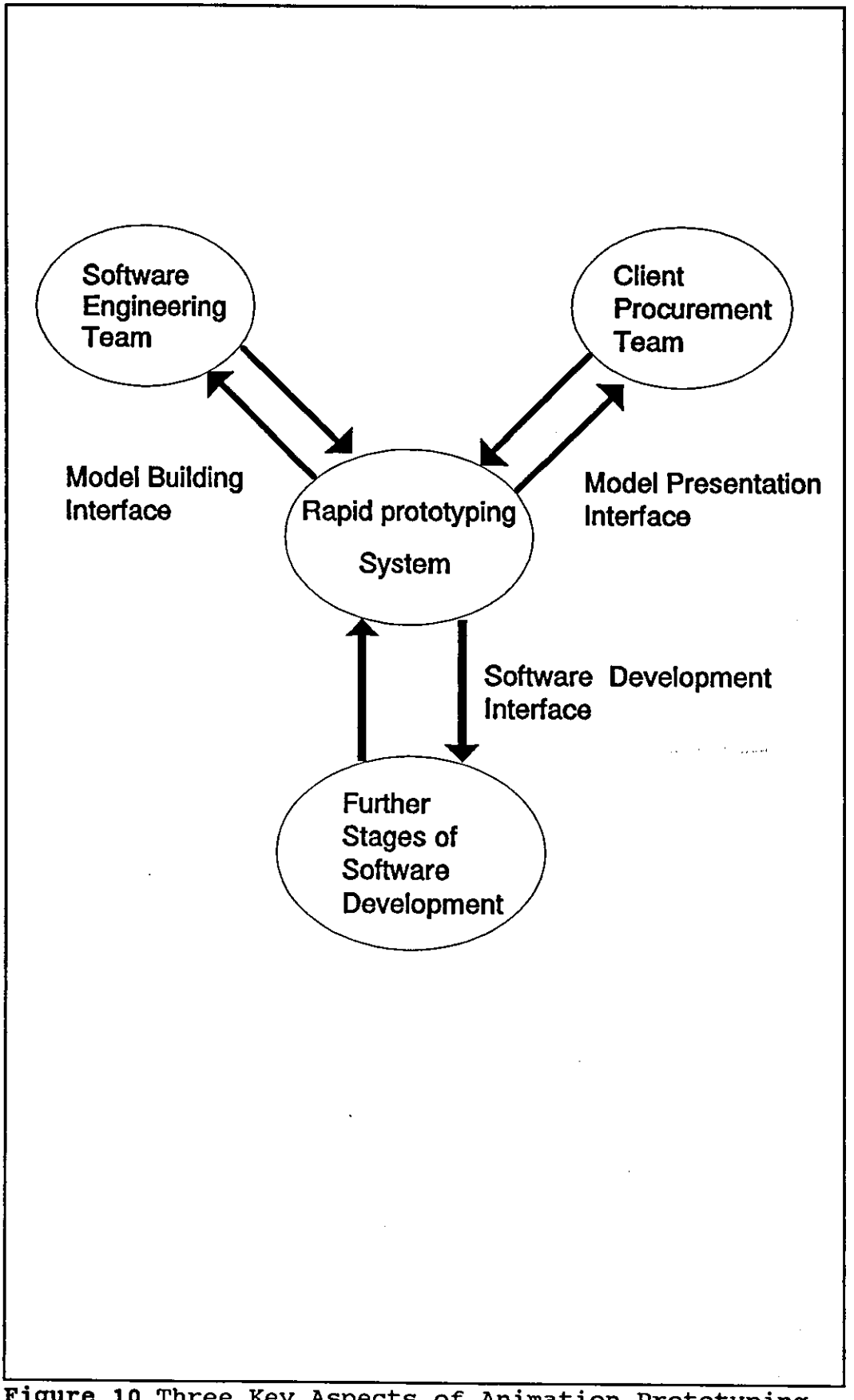


Figure 10 Three Key Aspects of Animation Prototyping.

number of issues:

(a) How can models be built quickly and economically?

(b) What sort of language should be used to build models?

  (i) Procedural.

  (ii) Declarative.

  (iii) Object-oriented.

  (iv) Graphical.

(c) How will the system support the modelling of the passage of time?

(d) Structured design will be needed to support the building of non-trivial models. What sort of technique should be adopted?

(e) Can development be speeded up by the reuse of certain components?

(f) A powerful user-friendly development environment will be needed. Can key features which would make such an environment particularly successful be identified? If so does such an environment exist? If not, is it possible to build such an environment?

(g) Support for the output of animated graphics will be needed. How can pictures be built quickly?

3.4.3 Using pictures - client-developer communications.

Another key issue is the use of animated pictures to show model properties to the customer. A number of important questions need to be answered.

(a) What type of pictures should be used?

(b) How do customers react to the use of animated pictures?

(c) How can key elements of a system be identified and subsequently animated?

(d) What sort of problems can be identified most easily by using pictures?

(e) What sort of problems are most difficult to identify using pictures?

These issues were clearly highlighted by the experimental work with custom-made prototypes described in Appendix A.

3.4.4 Using the results - onward into software design.

Once animation prototyping has been used there is a need to convert the findings into a form which can be used to guide the software design process. Some diagrammatic techniques can produce pseudo-code or executable code. Building prototypes using these techniques not only leads to better

prototypes, but also provides very useful results for the design of the software.

3.5 Animation Prototyping of Formal Specifications.

3.5.1 The basic concept.

The approach suggested is the automatic production of animated prototypes from formal specifications. As a response to the issues highlighted above, the major concern of this work has been to investigate a method for assessing the accuracy of system requirements as expressed in a formal specification. The use of formal specification techniques in commercial projects presents many advantages, but also presents some difficulties. Formal specifications allow the precise expression of ideas. However, the mathematically correct specification could actually describe a dangerous system. The correctness of a specification within itself is no guarantee that the actions described will not have dangerous consequences. The essential difficulty with formal specifications is the link between mathematical quantities and properties and their "real world" equivalents. The systematic approach taken here is shown in Figure 11. This approach address the three key issues of animation prototyping mentioned above in the following ways.

3.5.2 Model building.

As already discussed, an effective animation prototyping system is based on tools which aid the rapid construction of models. Such a tool must also provide the developer with a powerful means of expressing the problem structure. Formal methods and their specification languages provide such a means of expression. In safety-critical real-time systems, where producing a formal specification is becoming increasingly widespread, production of the prototype stems directly from work done on producing the specification. This makes animation prototyping by this means more attractive financially.

Automatically generated prototype code gives a consistent interpretation of formal specifications. This is an important aspect of this approach. If prototype code were produced manually, how can one be sure that the animation accurately illustrates what the formal specification means? With the increased use of automated animators, confidence in the translation algorithms can be increased. Ultimately full and rigorous proof of these algorithms will be possible.

3.5.3 Style of pictures and discussions.

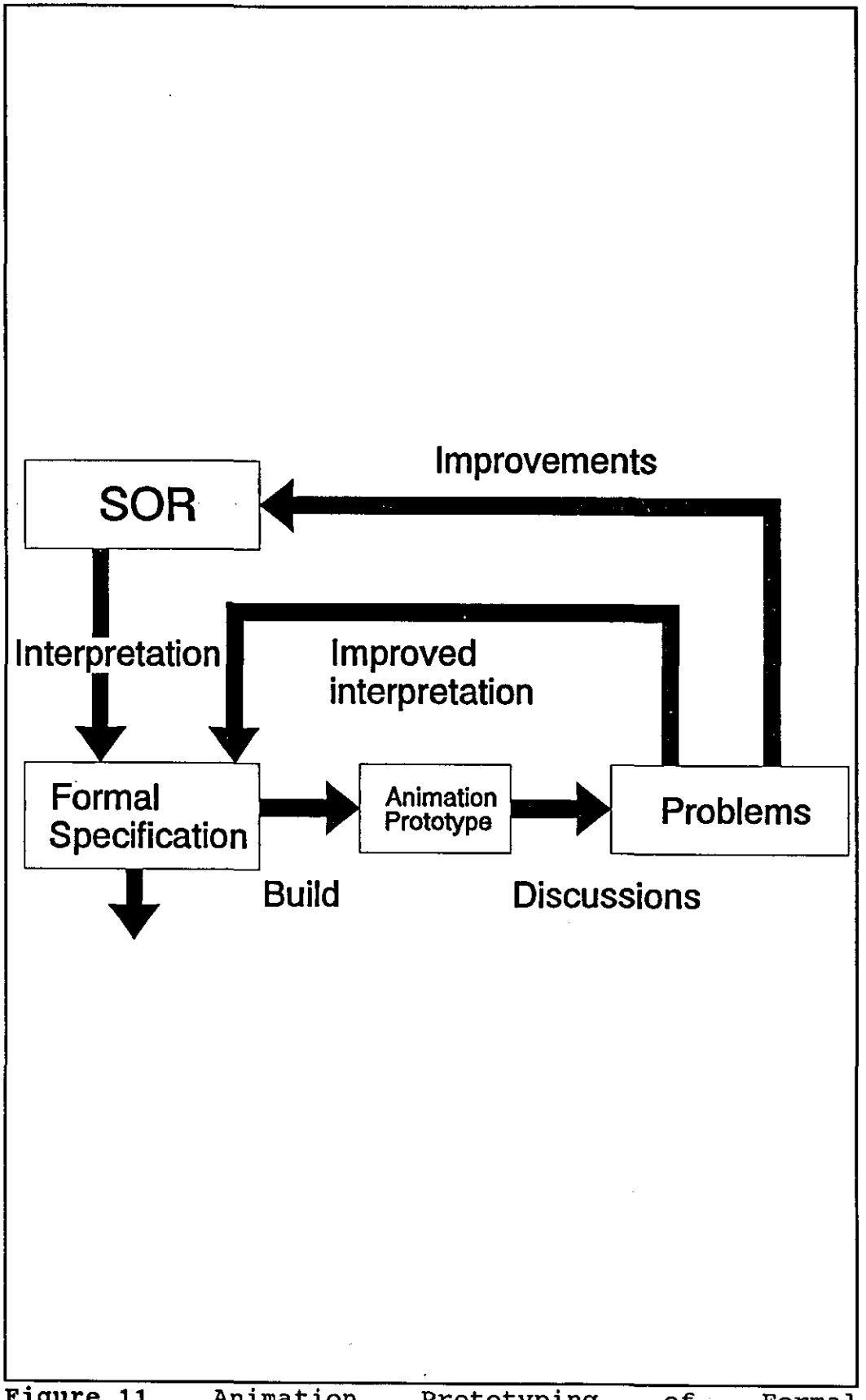


Figure 11 Animation Prototyping of Formal Specifications.

The interaction between client and developer is aimed at producing a greater understanding of the requirements as specified. The animations aid non-computer specialists in understanding the behaviour described by formal notations, as they are picture based. This in turn helps the developers in the production of a specification which accurately reflects the clients needs. The pictures used as a basis for the computer displays are diagrams which are familiar to the client. The examples of animation prototyping given in chapter 8 show this in practice.

The separation of model code from the pictures means that the pictures can be altered easily to give a clearer representation of the problem. With the system used changes to the look of the picture can be performed immediately and demonstrated without having to rework the model code.

3.5.4 Helping the development of software.

From the software developers viewpoint, the main advantage of animation prototyping directly from a formal specification is that the result of the prototyping exercise is a carefully constructed formal specification of the requirements. The formal specification acts as a sound basis for the subsequent development of the software.

As the software design is often undertaken by software engineers other than the specifiers, the animated prototype useful aid to understanding the specification and the application of the software.

Furthermore, in safety-critical applications, when the software has been coded, tools like SPADE and MALPAS can be used to verify the software against the formal specification.

3.6 Summary.

This chapter has given the background to software prototyping. The ideas of rapid prototyping and animation prototyping have also been introduced. Through the presentation of practical work attention has been drawn to key areas of concern for viable animation prototyping techniques. In response to these and other concerns raised in the previous chapter, the use of animation prototypes of formal specifications has been proposed. The next two chapters give a detailed discussion of what formal methods are and VDM, the method on which further practical work has been based, in particular. From this basis chapter 6 discusses the development of a practical approach to animation prototyping of formal specifications.

4 FORMAL SPECIFICATIONS (GENERAL)

4.1 Introduction.

Previous chapters show the need for tools and techniques to aid the specification process. A brief outline of the use of formal systems for software engineering is given in chapter 1. In this chapter the concept of formal specifications are discussed in more detail. The mathematical foundations of formal systems, the terminology and techniques used in building formal specifications are given in Appendix B. This chapter and subsequent ones assume a working knowledge of the concepts introduced there.

A number of the most popular, practical formal systems for software development are given. This chapter then highlights some of the benefits of using formal specifications and sets this against some of the more important drawbacks. Also given are the reasons for the choice of one particular system, called the Vienna Development Method, for practical work on building animation prototypes from formal specifications.

4.2 Practical Formal Systems for Software Engineering.

4.2.1 Different types of formal systems.

The phrase "formal systems" is used a great deal in mathematical text books on logic and reasoning. A more commonly encountered phrase, in the context of software engineering, is "formal methods". This is used to describe formal systems developed specifically for application to the specification and development of software. As will be seen these formal methods are not really "methods" at all; they do not prescribe a series of actions which if performed will produce a piece of software. Rather, they are tools which software developers may apply to help them develop software accurately from a specification.

To better understand the use of the mathematics in the construction of specifications in the most popular, current formal methods, the following classification is used:-

(a) Model-based systems.

(b) Algebraic systems.

(c) Process algebras.

(d) Temporal and modal logics.

These distinctions are not clear cut. The model-based systems can be used to build systems in an algebraic style and vice versa. However, systems have been classified according to their more usual method of employment.

4.2.2 Model-based systems.

The most widely used formal methods in this class are the Vienna Development Method (VDM) and Z (VDM is discussed in much greater detail in the next chapter).

The model-based (also called state-based) approach to specifications centers around a mathematical model which describes the state of the system in question. In addition to the state there are also operations on the state. The formal languages in these methods describe new data types in terms of fundamental mathematical entities such as Boolean, real, integer and natural numbers. More complex types can be built by using sets, maps and sequences. These types all have well understood properties which can be used to reason about data types defined in terms of them.

Having defined a state, operations which modify that state may be described. Operations are defined in the following terms:-

(a) input arguments.

(b) output arguments.

(c) influence on state variables.

(d) the relationship between the values of these variables under the action of the operation.

Specifications are thus formed from declarations of data types and definitions of operations on the state. Such specifications may be "refined" by making details of the data structures in the state closer to those available in the implementation language. This is called data reification. Alternatively, operations can be decomposed into simpler operations. This is called operation decomposition. With both of these techniques, decomposition and reification, "proof obligations" arise. That is, it is incumbent upon the refiner to demonstrate that the refined specification has the same properties as the original specification.

In order to demonstrate this style of specification (and others in later sections) in action, consider the following very simple specification of a LIFO stack. Informally the stack has the following properties:-

(a) The stack stores natural numbers.

(b) Items may be added to the stack.

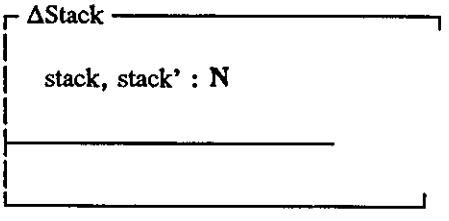
(c) The top item on the stack may be inspected.

(d) The top item may be removed from the stack.

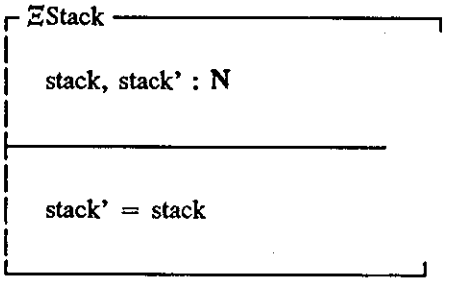
(e) Items are removed from the stack according to the last-in first-out principle.

As VDM is to be considered in the next chapter, Z will be used here to give demonstration of a contrasting style of writing model-oriented specifications. The Z notation was originally developed by the Programming Research Group at Oxford University, UK. Z is based on predicate calculus and set theory. It also has a powerful schema notation for organising specifications. This schema notation allows schema to be combined using operations such as extension, restriction, inclusion and composition. This notation is dealt with in greater detail in Spivey's book [Spivey89]. The use of the Z notation in the specification of systems can be found in Hayes' book [Hayes87].

The basic mathematical entity used to represent the stack is a sequence of natural numbers. The following schema defines the variable "stack".

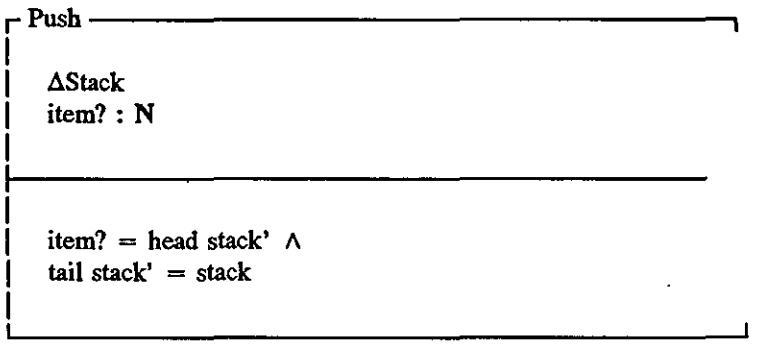


Three operations on the stack are to be considered. Operations in Z are defined in two parts. First is the signature (the part above the middle dividing line) which declares items of interest on which the operation depends. Second is the predicate (the part below the dividing line) which describes the logical relationship between those items. The next schema is defined for inclusion in those operations which do not change the state. It simply says that the stack before the operation is the same as the stack after the operation.

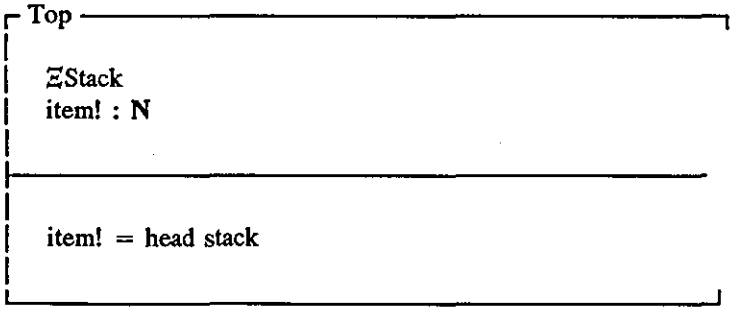


The Push operation places a single item on the top of the stack. It has two lines in the signature. The

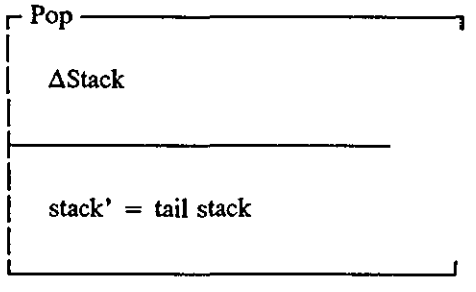
first "ΔStack" declares that this schema is included in the new schema. The predicate, none in this case, is conjoined with the new predicate. The second "item?" declares that there is an input argument of type natural number. The predicate part states that the input is equal to the first item in the stack (the "head") after the operation and that the rest of the stack (the "tail") after the operation is equal to the stack before the operation.



The second operation, Top, returns the value of the item at the top of the stack. The signature is similar to that for Push, only now the line "≡Stack" includes that schema and hence Top does not change the state. The line "item!" declares that there is an output argument of type natural number. The predicate part simply states that the output is equal to the head of the stack before the operation.



The third operation, Pop, removes a single item from the top of the stack. It has no input or output arguments, but it does change the state by making the stack after the operation equal to the tail of the stack before the operation.



This example serves to give a flavour of model-oriented specifications and helps to show the similarities and contrasts between the Z and VDM notations. The most striking difference between them is the lack of separation of pre- and post-conditions in Z. However, the logical expressions for pre- and post-conditions may be derived. This calculation is essentially the reverse of the implementability proof in VDM.

4.2.3 Algebraic or axiom-based systems.

These systems are based on the representation of a system as sets of data types and equations describing the properties of those data types. In building an algebraic specification of a data type theories about the data type are constructed. These theories, in contrast to the model-based approach above, do not generally introduce general mathematical objects for the purposes of modelling. Instead they rely only on the underlying logical system for presentation of their theories.

In order to see how this approach works consider again the simple stack specification again. In order to specify this system a "theory of stacks" is built. Once again the stack consists of natural numbers. In algebraic specifications one builds a specification by "enriching" or "importing" (depending on the particular notation) theories about more fundamental data types. Theories of data types typically consist of two main parts. Firstly, a list of the signatures of all the operations on the data type. Secondly, a list of axioms describing the relationship between those operations. Also additional information specific to the notation may be given.

Here, for the sake of simplicity, a very simple algebraic notation is used. This specification consists of five parts. These are:

(a) A name for the specification and details of any other specifications which are required.

DATATYPE stack **ENRICHMENT OF** natural

(b) A definition of the sort (or type) of the entities being described.

SORTS STACK

(c) A signature part where operations are given names and the sorts of their parameters are defined.

OPERATIONS:

Top : STACK → NAT
Pop : STACK → STACK
Push : STACK × NAT → STACK

(d) A variable part where variables used in the axioms part are named and their sorts are given.

VARIABLES:

item : NAT
st : STACK

(e) An axioms part where the relationships between the sort's operations are defined.

AXIOMS:

Top(Push(st,item)) = item
Pop(Push(st,item)) = st

The interpretation and meaning of this are not obvious. However, the axioms in the system allow reasoning not only about truth values of statements, but also about the values of objects. It is possible to substitute equal expressions into other expressions. This is referred to as rewriting expressions and consequently, as they define equal terms, the axioms are referee to as rewrite rules. Consider the following expression:-

Top(Pop(Push(Push(X,10),4)) . . . . . (1)

where X is some arbitrary stack. To answer the question "what value does Top return under this sequence of operations?", refer to the above stack specification. It can be seen that:-

Pop(Push(st,item)) = st. . . . . (2)

The Push operation returns a value of sort STACK. If the substitution st = Push(X,10) is used in (2) the expression (1) can be rewritten as:-

Top(Push(X,10)) . . . . . (3)

From the axiom:-

$$Top(Push(st,item)) = item \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (4)$$

It can be seen that expression (3) can be rewritten as:-

$$10 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (5)$$

This answer can be verified operationally by hand.

Practical formal methods using this style of specification are Clear [Burstall80], ACT ONE [Ehrig85], OBJ [Goguen79] and Larch [Guttag85 Guttag86]. These formal methods essentially contain theories of theories. That is they are based on mathematical constructs which encompass the building and re-use of theories about data types. From the point of view of animation all these methods are interesting. Those familiar with functional programming languages such as PROLOG, LISP and ML will see the possibilities of translating the axioms into rules and functions in those languages. OBJ is the most advanced in this respect, having a toolkit called ObjEx [ObjEx90] which is, as the name suggests, an executable subset of OBJ and tools to support the construction of specifications.

The biggest drawback of this style of specification, particularly with large systems, is that the axioms are difficult to construct. Maintaining the consistency of the axioms in a large specification is difficult. As Goguen and Tardo [Goguen79] have pointed out:-

> "The sad fact is that a disturbingly large number of published algebraic specifications, even of simple data types, are wrong."

4.2.4 Specifying concurrent systems - Process Algebras.

Research into concurrency has proved to be a breeding ground for formal systems. The major formal systems to emerge form this are a Calculus of Communicating Systems (CCS) [Milner80] and Communicating Sequential Processes (CSP) [Hoare85]. The communication protocol specification system LOTOS [Brinksma88] is based on CCS and ACT ONE. The process algebras describe the behaviour of systems by defining their algebra. A system's algebra specifies the components of the system (agents in CCS and processes in CSP) and what the observed behaviours of those components will be. The observed behaviour of a component is called its trace.

In the CSP notation "processes" which perform "actions" are specified. Also, CSP includes the notion

of traces which are the observed behaviour of processes, i.e. a sequence of the names of all events in which the process participates. Using the logical calculus combinations of processes, actions and communications channels can be analysed for deadlock and other useful properties. Consider, once again, the specification of a simple stack. This time the stack will be specified as a process.

$$STACK = P_{\langle\rangle}$$

where $P_{\langle\rangle} = (input?x \rightarrow P_{\langle x \rangle})$

and $P_{\langle x \rangle \hat{} s} = (output!x \rightarrow P_s \mid input?y \rightarrow P_{\langle x \rangle \hat{} \langle y \rangle \hat{} s})$

The first line defines the STACK to behave like the process P behaves on an empty trace. The behaviour of P in this case is to consume an item "x" from its input channel and then behave like P after the single event trace x. The last line says that P after some arbitrary trace s followed by event x can do two things. It can output x and remove it from the stack, i.e. it subsequently behaves like P after the arbitrary trace s. Alternatively it can consume an item from its input channel and add it to the stack, i.e. it behaves like P after the arbitrary trace s followed by the input event x followed by the input event y. As with the algebraic specification above this definition is recursive.

In particular, the similarity between the observed behaviour of two specifications can be deduced. If P is some product which meets specification S, i.e. P satisfies S or P sat S. Every set of observations of the behaviour of P is described by S. Formally this is :-

$$\forall tr.tr \in traces(P) \rightarrow S$$

where traces(P) is the set of possible observed behaviours of P. CSP includes a language for describing traces and a deductive apparatus for reasoning about traces and relationships between traces.

4.2.5 Temporal and modal logics

Temporal, or modal, logic is a property-oriented method for specifying the properties of concurrent or distributed systems. There are no standard temporal inference systems nor are there standard temporal operators. However commonly encountered concepts are those of "always", "eventually" and "next". Unlike classical logic, the truth of temporal logic predicates may change with time. Consider a predicate P with respect to some sequence of states, the following statements:-

$\Box P$

$\Diamond P$

$\bigcirc P$

can be stated informally as P is true in all future states (always), P holds in some future state (eventually) and P is true in the next state (next), respectively. Temporal logic specifications are an unstructured list of predicates which must be satisfied by a given implementation. Yet again consider the simple stack specification using a temporal logic.

$$\langle \text{output!m} \rangle \Rightarrow \blacklozenge \langle \text{input!m} \rangle$$

$$(\langle \text{output!m} \rangle \wedge \ominus \blacklozenge \langle \text{output!m'} \rangle) \Rightarrow \blacklozenge (\langle \text{input!m'} \rangle \wedge \ominus \blacklozenge \langle \text{input!m} \rangle)$$

$$(\langle \text{input!m} \rangle \wedge \ominus \blacklozenge \langle \text{input!m'} \rangle) \Rightarrow (\text{m} \neq \text{m'})$$

$$\langle \text{input!m} \rangle \Rightarrow \Diamond \langle \text{output!m} \rangle$$

These four predicates may be paraphrased informally. The first predicate states that any message transmitted to the "output" channel must have previously been placed on the "input" channel. The second predicate captures the requirement for the last in first out behaviour of the stack. If a message m placed on the output channel is preceded by some other message m' also on the output channel, there must have been a preceding sequence of events consisting of placing m' on the input channel preceded by an event that placed m on the input channel. The third predicate states that all messages are unique. The last predicate deals with the liveness of the system. It states that each incoming message will eventually be transmitted.

The FOREST Alvey project produced a more mature modal logic called Modal Action Logic or MAL [Maibaum86]. In MAL a specification consists of two parts, a declarations part and an axioms part. The key concepts of the modal logic are those of the sorts agent (Ag) and action (Ac).

Modal formulae can be formed as follows:-

If $A \in Ag$, $a \in Ac$ and α is a formula, then

$$[A,a]\alpha$$

is a formula.

This means that A is a member of the set of all agents (Ag) and a is a member of the set of all actions (Ac). The formula should be read as; if agent A does action a then in the resulting state α holds. These formulae may be used to build formulae using classical logical connectives, e.g.

Pre → [A,a]Post

This can be read as Pre is a true formula before agent A performs action a and Post is a true formula afterwards.

Returning to the simple stack example, the following specification can be constructed:-

Data Sorts: STACK, NATURAL;

Variables:

item : NATURAL;
st : STACK;

Agents:

A, B

Actions:

PUSH : NATURAL × STACK;
POP : STACK;

Predicates:

Top ⊆ STACK × NAT

Axioms:

[*A*, *PUSH*(st,item)](*Top*(st,item))
$F \rightarrow$ [*A*, *PUSH*(st,item)][*B*, *POP*(st)]*F*

In particular:

Top(st,item)
→ [*A*,*PUSH*(st,item)][*B*,*POP*(st)]*Top*(st,item)

Most of this is familiar except for the Axioms section. The first axiom states that if some (arbitrary) agent A performs the action PUSH using the parameters st and item then the predicate top(st, item) will be true. The second axiom says that if a formula is true then after A performs PUSH and B performs POP the same formula will be true. The most useful case of this is given; that the top of the stack is unchanged after a single PUSH followed by a single POP action.

MAL also has the so called "deontic" operators per, obl and ref. These deontic operators have two arguments an agent and an action. They are informally stated as follows:-

* per. The agent is permitted to perform the action.
* obl. The agent is obliged to perform the action
* ref. The agent is permitted to refrain from performing the action.

Using these operators, axioms can be built describing the complex inter-relationship between agents and sequences of events.

4.3 Advantages and Drawbacks of Formal Systems.

4.3.1 Advantages of formal methods.

The use of formal methods is seen as having many potential benefits. The positive aspects have been represented diversely by the many writers on the topics.

Briefly stated [Jones90], the two main arguments in favour of formal methods are:-

(a) The use of mathematics provides precision and brevity.
(b) Designs can be verified. The relationship between a program and its specification is open to formal reasoning.

Further to these [Bloomfield88, Floyd85, Hall90], formal methods are seen as being useful because:-

(a) Formal methods are very helpful at finding errors early on and can nearly eliminate certain classes of errors.
(b) The process of formalisation in the early stages of software development will promote the finding of errors, inconsistencies and missing elements in the requirements. Formal methods

work largely by making one think very hard about the system you propose to build.

(c) On the basis of the specification, software developers will be able to answer questions on the intended functionality of the program at a very early stage.

(d) Programs can be proved to be correct with respect to their specifications. As they contain a methodological framework within which software may be developed from the specification in a formally verifiable manner.

(e) Specifications can be transformed into programs according to fixed rules ("mechanically").

(f) In some methods, specifications can be executed and thus serve as a prototype for the future system.

(g) They are useful for almost any application.

(h) They are based on mathematical specifications, which are much easier to understand than programs.

(i) They can decrease the cost of development.

(j) They can help clients understand what they are buying.

(k) They are being used successfully on practical projects in industry.

In summary formal methods, through their use of mathematics, offer developers a precise and concise mechanism for specifying systems. By focusing effort on clearly expressing the specification, they give developers a better understanding of the nature of the problems being addressed. Through successive stages a specification can be further refined in a way which verifiably preserves properties of the original specification. The use of a systematic approach to software development can produce software which can be checked against its specification by a process of formal reasoning.

4.3.2 Drawbacks of formal methods.

There are many criticisms which have been levelled at formal methods [Hall90]. The more serious criticisms, for software engineers, are as follows:-

(a) Proofs are very large and the mathematical manipulations are involved.

(b) Specifications are incomprehensible to non-specialists.

Although formal proofs have been suggested as one of the main reasons for using formal methods, fully formal proofs have yet to enter into widespread usage in the development process. There are two main reasons for the lack of formal proofs. Firstly, proofs are reasoned arguments. There is nothing formal or mechanical about the construction of a proof. A proof must be carefully built and revised repeatedly until it too is correct. The notations are tedious to manipulate, consequently this is a time-consuming and error-prone process, even with small proofs. Secondly, proofs of real systems are very

large and involved. In text books, the specifications one sees are written with proof in mind. The resultant proofs are elegant and compact. With specifications of systems built with incomplete information about the whole system, the proof of properties will not flow so neatly.

It has been suggested [Hall90] that many of the benefits of formal methods are attainable with little or no use of formal reasoning. For complex systems development, this could seriously detract from the appeal of formal methods. Much work has centered around issues such as proofs of safety and timing properties and it is these issues which have aroused the most interest in formal methods [DEF STAN 00-55]. Without the support of proofs formal methods are considerably weakened. There are a large number of published specifications which have errors in them which would have been highlighted by performing the necessary proofs. In the absence of proofs formal methods could become little more than very sophisticated specification languages.

The second and more significant drawback in the use of formal methods for the capture of system requirements is their use of specialised mathematics. While this provides developers with useful support for their construction of the specification it makes the specification inaccessible to non-specialists. As has already been argued, the building of high quality specifications (formal or otherwise) relies on discussions about what the specification says. This lack of accessibility is one of the main reasons for advocating the use of animation prototypes of formal specifications.

4.4 The Final Choice of Formal Method for Animation Prototyping.

The formal method chosen as the basis for the practical work demonstrated here was VDM. The main requirements were that the method:-

(a) be mature and well-documented. The primary interest of this work was to explore the animation of formal specifications and not to address language issues.

(b) have constructs to deal with the ordering of events in time. The specification of real-time embedded systems is also central to this work.

(c) support a structured approach to specification building. Specifications may be built by a team of people working together. There is thus a need to partition a problem into manageable pieces.

The first requirement narrowed the field to a choice of five notations; VDM, Z, OBJ, CCS and CSP. Languages such as LOTOS and MAL are not really in sufficiently wide usage, in the academic

community, nor was sufficient documentation available. The second requirement would appear to make CCS and CSP the obvious choices, but these notations are only for describing communication between processes. They are not suitable for describing the internal structure of those processes. Ultimately, then, the choice was between OBJ, VDM and Z. Subsets of OBJ are executable, however, its algebraic specification style and, in particular, the term rewriting approach to execution, is not well suited to the animation prototyping of real-time systems. VDM was chosen, in preference to Z, because of work already carried out in the department and by the industrial collaborators at Rolls Royce and Associates.

5 SPECIFYING SYSTEM REQUIREMENTS USING VDM.

5.1 Origins of VDM and Current Research.

The ideas of formal specification and rigorous software development methods have their origins in the 1950's and 60's. Work at IBM's Vienna research laboratory on the PL/1 programming language produced a formal specification of the language. This specification was based on "operational semantics" using an approach called the "Vienna Definition Language" [Lucas69]. The need for mathematical specification techniques and the basic theoretical mechanisms were firmly established by 1970. With this increased understanding of the more powerful specification techniques of "denotational semantics", the VDL approach was improved and became known as META-IV. This meta-language was incorporated with the concept of systematic software development and together this approach formed the Vienna Development Method or VDM. Bjorner and Jones [Bjorner82] give a thorough treatment of the theoretical basis of the meta-language. They also gives example of programming language specification and program development using VDM. Since then VDM has been further refined as the theoretical understanding of its mathematical foundations have improved [Bjorner87, Bloomfield88, Jones90]. The latest developments have occured in the syntax of META-IV. Many of the new ideas have arisen from the proposed British Standards Institute standard for VDM [BSI89].

5.2 Mathematical Foundations.

The intention of VDM is to allow a designer to transform a specification written in an implicit form into a program which can be implemented in a programming language. The process of transition from an initial formal specification to an implementation is called stepwise refinement. In this approach, additional information and design descisions are made at each step (see Figure 12). This produces a new refined specification. Mathematical checks can be made to see if the important properties of the previous specification have been preserved and incorporated into the new specification. This chapter and its associated appendix C introduce the concepts of specification and verifiable stepwise refinement in VDM.

Like the Z specification language discussed in the previous chapter, VDM uses a model-oriented approach to specification. Thus specifications in VDM have a state which models the system being specified. It also has a number of operations which influence that state, these are used to specify the

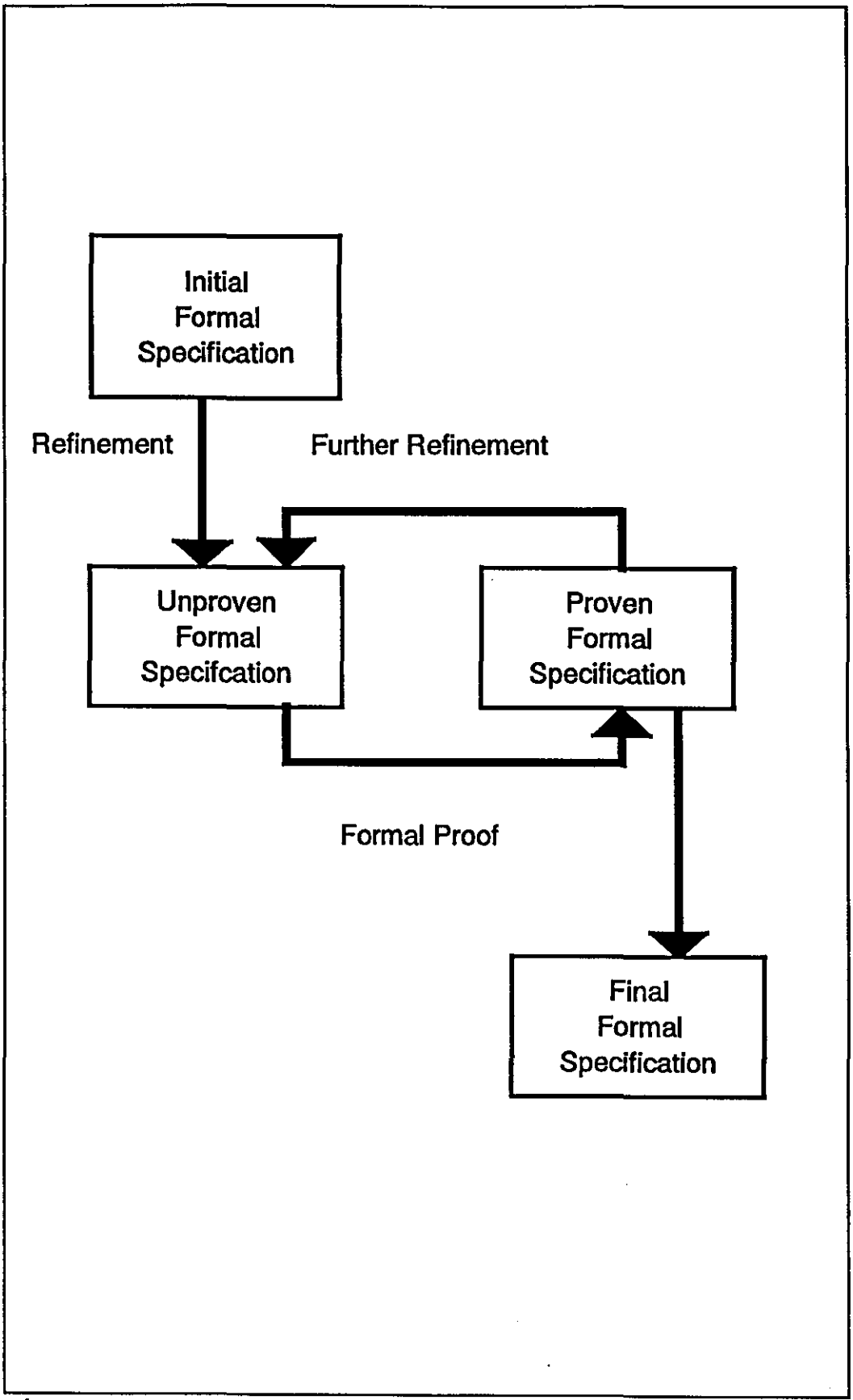


Figure 12 Stepwise Refinement of a Formal Specification.

functions performed by the system. There are two mathematical cornerstones underpinning VDM, these being:-

(a) The formal mathematical system of first order predicate calculus, which is an extension of propositional calculus.

(b) The extension to (a) known as the logic of partial functions, or LPF.

A detailed treatment of the mathematics involved is given by Bjorner and Jones [Bjorner82]. This also includes an extensive list of references to earlier mathematical texts from which the VDM approach has developed. Jones [Jones90] gives a much abridged, but more accessible, introduction to these subjects. It is sufficient here to recognise that the abstract syntax of the META-IV language has been soundly established and that, on this basis, a series of inference rules have been formed. These inference rules allow the developement of software from META-IV specification to be supported by verification using formal proofs.

5.3 VDM Specifications - describing basic and composite data types.

VDM specifications are built by modelling a system in terms of basic mathematical entities such as Boolean variables (B) and Natural numbers (N). In addition to these VDM includes three further entities. These are finite sets, finite maps and finite sequences. Specific instances of the above three entities may be defined in two ways, enumeration and comprehension, e.g.

enumeration:

$$s = \{ 1, 2, 4 \}$$

comprehension:

$$squares = \{ x \mapsto x^2 \in \mathbf{N} \times \mathbf{N} \mid x \in \{ i \in \mathbf{N} \mid -2 \leq i \leq 3 \} \}$$

i.e. $square = \{ -2 \mapsto 4, -1 \mapsto 1, 0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9 \}$

The full details of how to define basic entities are given in appendix C.

These simple entities may be used to describe simple data types to be used in a specification. Further to this VDM has a mechanism for combining simpler entities together to form composite objects. In many respects these composite objects are like Pascal records. Composite objects have a number of fields, of specified types, and tags (names) for each of those fields. For instance a composite type for a buffer of natural numbers of defined size may be defined as:-

compose *Buffer* of

Max-size : N_1,

Store : N^*

end

Composite objects are most frequently used in the specifications of state variables and so names may be associated with the set of composite objects defined:-

Finite_Buffer = compose *Buffer* of

Max-size : N_1

Store : N^*

end

The definition of *Buffer* is not yet complete as the size of the *Store* field has not been constrained. It is necessary to restrict the possible combinations of values taken by this data type. This is achieved by the use of a "data type invariant". This is a truth-valued function which is true for all valid object of the type. Invariants are defined as follows:-

inv-Buffer(*Buf*) $\triangleq$ len *Store*(*Buf*) $\leq$ *Max-size*(*Buf*)

This says that for valid objects of type *Buffer* the number of elements in the *Store* sequence is always less than or equal to the *Max-size* of the buffer. The details of how composite objects are defined and manipulated are shown in Appendix C. With the basic data definition mechanisms established, definitions of operations and states can now be considered.

5.4 VDM Specifications - functions, operations and states.

5.4.1 Explicit definitions of functions.

Functions define a fixed mapping from input to output. For any function it is useful to record its domain (the set of values to which a function may be applied) and its range (the set which contains the results of the functions application). This is called the function's signature. A function's signature is written with domain and range sets separated by an arrow. The signature of a function to double a given number might be:-

double: $\mathbf{N} \rightarrow \mathbf{N}$

For functions which take more than one argument, the domain is written as the appropriate sets separated by crosses. For example:-

modulo: $\mathbf{N} \times \mathbf{N}_1 \rightarrow \mathbf{N}$

Functions can de defined as expressions with terms of other already understood or defined functions. *Double* can be defined directly as:

$double(i) \triangleq 2 * i$

The symbol $\triangleq$ means "is defined as". It is used to distinguish from the = symbol and its meaning notion of equality. The *modulo* function can be given the direct definition:-

$modulo(i,j) \triangleq i - i \text{ div } j$

The result of this operation is bound to the value of the expression $i - i \text{ div } j$; where div is integer division.

5.4.2 Implicit definitions of functions.

Direct definition of functions is restrictive when building specifications as too much emphasis is placed on how a calculation is to be performed. Implicit definition of functions allows one to concentrate on what a function should achieve. The general format for such definitions is:-

function (*input* : *input_type*) *output* : *output_type*
pre ... *input* ...
post ... *input* ... *output* ...

The first line gives the signature of the function in a format more recognisable to those familiar with languages such as Pascal. The pre section or pre-condition is a truth-valued function (*pre-function*) defining assumptions about the arguments. It has the signature:

pre-function : *input* $\rightarrow \mathbf{B}$

The post section or post-condition defines a relationship between the input and output parameters (*post-*

function). It has the signature:

$$post\text{-}function : input \times output \rightarrow \mathbf{B}$$

It is important to note that truth of the post-condition expression is conditional upon the truth of the pre-condition. Formally the relationship is:

$$\forall\, input \in input\_type \bullet pre\text{-}function(input) \Rightarrow \exists output \in output\_type \bullet post\text{-}function(input, output)$$

This may be paraphrased as say for all input of the correct type which satisfy the pre-condition there exists at least one output of the correct type which with the input satisfies the post-condition. For input values which do not meet the pre-condition the truth of the post-condition is undefined i.e. the value of the output is not constrained.

As an example of the merits of implicit specification consider a function which is given a set of natural numbers and returns the lowest number in that set.

$$Minimum(in\_set : \text{N-set})\ lowest : \mathbf{N}$$
$$\text{pre } in\_set \neq \{\}$$
$$\text{post } lowest \in in\_set \wedge \forall\, i \in in\_set \bullet lowest \leq i$$

This specification has many advantages. All the required properties are expressed without needing to detail how such a function might be implemented. The pre-condition states that the input must not be an empty set. The post-condition states that the returned value must be a member of the input set and that it must be less than or equal to all other members of that set. Note that for empty sets the function is undefined.

Unless one is an adherent of a strictly functional style of programming, the idea of specifying complex computer programs solely in terms of functions is unfamiliar. Of much greater usefulness is the concept of states and operations which act on states.

5.4.3 States and Operation definitions.

Operations are functions whose applications affect and are affected by a state. Functions define a fixed mapping from input to output. The application of the function *double* yields 4 when applied to 2 regardless of whether it had previously been applied to other values or not. Operations, however, have

a hidden state which is used to record values which affect subsequent results. For example, an accumulator might respond to an input of 2 with 2, to 10 with 12 and to a second 2 with 14. The state of an operation is a collection of external variables which it can access and change.

The state of an operation or collection of operations can be defined as a composite object. Consider the buffer object refered to above. This object could be used as a state as follows:-

state

State of *Buf* : *Buffer*

init (*mk-State*(Buf_0)) $\triangleq$ Buf_0 = *mk-Buffer*(256, [])

end

This defines the state to be called *Buf* and to be of type *Buffer*. It also defines an initial state for the buffer Buf_0, where the *Max-size* is 256 and the *Store* is an empty sequence.

Operations can now be defined which act on the state. The general format of such operations is :-

OP (i : T_i) o : T_o
ext rd v_1 : T_1
wr v_2 : T_2
pre ... i ... v_1 ... v_2 ...
post ... i ... v_1 ... $\underline{v_2}$ ... o ... v_2 ...

Most of this is familiar from the implicit definition of functions. However the two lines of the external clause (keyword "ext") need explanation. The first line states that the operation has read only access to the state variable v_1 of type T_1 i.e. it may use but not change that value. The second line states that the operation has read and write access to the state variable v_2 of type T_2 i.e. it may use and change that value. The pre-condition now has a signature:-

$pre\text{-}OP : T_i \times T_1 \times T_2 \rightarrow \mathbf{B}$

The post condition now has to deal with variables whose final values differ from their initial values. This is done by "decorating" the variable name to indicate initial values (This is conventionally done with a hook or $\leftharpoondown$. However, this is very difficult to achieve typographically. Therefore the convention of underlining initial values is adopted here, e.g. $\underline{v_2}$.) The signature of the post-condition thus becomes:-

$$post\text{-}OP : T_i \times T_1 \times T_2 \times T_o \times \underline{T_2} \rightarrow \mathbf{B}$$

This can be demonstrated by considering an operation which adds an item to the buffer defined above:-

Add_item (*item* : **N**)
ext rd *Max-size(Buf)* : $\mathbf{N_1}$
  wr *Store(Buf)* : $\mathbf{N}^*$
pre len *Store(Buf)* < *Max-Size(Buf)*
post *Store(Buf)* = $\underline{Store(Buf)}$ ⌢ [*item*]

This operation is quite straightforward. The pre-condition states that it is only defined when there is room in the store for at least one more item. The post-condition states that the store after the operation is equal to the store before the operation concatenated with a sequence containing the item. This operation could be redefined to deal with the case when the buffer is full. The specifier may choose either to discard the new item or lose the item at the other end of the buffer. The latter case will be shown here.

Add_item (*item* : **N**)
ext rd *Max-size(Buf)* : $\mathbf{N_1}$
  wr *Store(Buf)* : $\mathbf{N}^*$
pre true
post (len $\underline{Store(Buf)}$ < *Max-Size(Buf)* ∧
  Store(Buf) = $\underline{Store(Buf)}$ ⌢ [*item*]) ∨
 (len $\underline{Store(Buf)}$ = *Max-Size(Buf)* ∧
  Store(Buf) = tail($\underline{Store(Buf)}$) ⌢ [*item*])

Now the operation has a pre-condition which is always true i.e. it is valid for all input values. The post-condition is now a more complex logical expression. It has two mutually exclusive disjuncts to deal with the two cases to be consider; when the buffer is full and when it has room. The first disjunct is the same as the previous operation. The second disjunct states that if the buffer is full then the store after the operation is equal to the tail of the store sequence before the operation (i.e. the sequence minus the first element) concatenated with the new item.

5.4.4 Proofs about states and operations.

Up until now the full use of the formality of VDM has not been introduced. Consider the question, how do we prove that what has been written makes sense? The first thing to state is that a formal

specification can not be linked formally to user requirements. However, the inference rules of VDM can be used to show that defined operations are implementable, this is called a satisfiablilty or implementability proof. Briefly, an implementability proof is a formal demonstration that for all valid inputs and initial states there exists some outputs and final states which are valid. The formal statement of this is:-

if σ is the state variable and Σ is its type

$$\forall \underline{\sigma} \in \Sigma \cdot pre\text{-}OP(\underline{\sigma}) \Rightarrow \exists \sigma \in \Sigma \cdot post\text{-}OP(\underline{\sigma}, \sigma)$$

For the *Add_item* operation, after an expansion of the relavent quantities as shown in appendix C, this is rewritten as:-

$$\begin{aligned}
&\forall \underline{Store(Buf)} \in \mathbf{N}^*, Max\text{-}size(Buf) \in \mathbf{N}_1, item \in \mathbf{N} \cdot \\
&\quad \text{true} \Rightarrow \exists Store(Buf) \in \mathbf{N}^* \cdot \\
&\quad (\ \text{len } \underline{Store(Buf)} < Max\text{-}size(Buf) \ \wedge \\
&\qquad Store(Buf) = \underline{Store(Buf)} \frown [item] \ \wedge \\
&\qquad \text{len } Store(Buf) \leq Max\text{-}size(Buf)\ ) \ \vee \\
&\quad (\ \text{len } \underline{Store(Buf)} = Max\text{-}size(Buf) \ \wedge \\
&\qquad Store(Buf) = \text{tail}(\underline{Store(Buf)}) \frown [item] \ \wedge \\
&\qquad \text{len } Store(Buf) \leq Max\text{-}size(Buf)\ )
\end{aligned}$$

The proof will take the form of a sequent:-

$$\begin{aligned}
&\text{from } \underline{Store(Buf)} \in \mathbf{N}^*, Max\text{-}size(Buf) \in \mathbf{N}_1, item \in \mathbf{N} \vdash \\
&\quad \text{true} \Rightarrow \exists Store(Buf) \in \mathbf{N}^* \cdot \\
&\quad (\ \text{len } \underline{Store(Buf)} < Max\text{-}size(Buf) \ \wedge \\
&\qquad Store(Buf) = \underline{Store(Buf)} \frown [item] \ \wedge \\
&\qquad \text{len } Store(Buf) \leq Max\text{-}size(Buf)\ ) \ \vee \\
&\quad (\ \text{len } \underline{Store(Buf)} = Max\text{-}size(Buf) \ \wedge \\
&\qquad Store(Buf) = \text{tail}(\underline{Store(Buf)}) \frown [item] \ \wedge \\
&\qquad \text{len } Store(Buf) \leq Max\text{-}size(Buf)\ )
\end{aligned}$$

$$\begin{aligned}
&\text{infer } \forall \underline{Store(Buf)} \in \mathbf{N}^*, Max\text{-}size(Buf) \in \mathbf{N}_1, item \in \mathbf{N} \cdot \\
&\quad \text{true} \Rightarrow \exists Store(Buf) \in \mathbf{N}^* \cdot \\
&\quad (\ \text{len } \underline{Store(Buf)} < Max\text{-}size(Buf) \ \wedge
\end{aligned}$$

$$
\begin{aligned}
&\mathit{Store(Buf)} = \underline{\mathit{Store(Buf)}} \frown [\mathit{item}] \wedge \\
&\text{len } \mathit{Store(Buf)} \leq \mathit{Max\text{-}size(Buf)}) \vee \\
&(\text{len } \underline{\mathit{Store(Buf)}} = \mathit{Max\text{-}size(Buf)} \wedge \\
&\quad \mathit{Store(Buf)} = \text{tail}(\underline{\mathit{Store(Buf)}}) \frown [\mathit{item}] \wedge \\
&\text{len } \mathit{Store(Buf)} \leq \mathit{Max\text{-}size(Buf)}) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \forall\text{-I(1)}
\end{aligned}
$$

This proof is shown in full and explained in appendix C.

This proof demonstrates a number of properties of the operation *Add_item* as specified:-

(a) There are some valid inputs which satisfy the pre-condition. In this case the pre-condition is trivial i.e. always true, but it is possible to write pre-conditions which when combined with data type invariants have no valid solutions.

(b) For all valid inputs there are some valid outputs which satisfy the post-condition. It is possible (or probable in complex specifications) that the post-condition expression conjoined with the data type invariant may be inconsistent i.e. it is always false, or be a contingency i.e. it is insoluble for certain valid inputs. An example of this can be seen if in the definition of *Buffer*, *Max-size* is defined to be of type N instead of N_1. In this case, the size of the buffer can be defined as 0 and it is impossible to apply the operation *Add_item* to a buffer of 0 size: the result of the concatenation is always of length $\geq$ 1 which violates the invariant len *Store(Buf)* $\leq$ *Max-size(Buf)*.

These sections have shown how using the basic mathematical entities of VDM, composite objects can be built to describe more complex data structures. The definition of functions which act on these data types has been shown in both a direct, or explicit, and implicit styles. The notion of state and the definition of such states has been covered along with the concept of operations which affect and are affected by the state have been discussed. Such ideas are suitable for definitions of data types and operations to manipulate them. However real-time systems deal with sequences of events which act to perform specific tasks. The next section looks at how such sequences of events can be described using VDM.

5.5 Building and Refining a Specification.

5.5.1 Operation decomposition.

In order to specify a real-time system it is necessary to set out in what order and under what conditions events occur. This is achieved by decomposing complex operations into combinations of simpler operations. The aim here is to implement systems using procedural languages, consequently control constructs of sequence, selection and iteration are used to link simple operation together. The decomposition of operations is supported formally by proofs that the decomposition is consistent with the definition of the operation being decomposed.

Consider the informal specification of a plant controller given in appendix C. In order to specify this system using VDM, the state is specified as follows:-

$valve = \{ \text{OPEN}, \text{CLOSED} \}$

state *Plant* of
    inlet : *valve*
    outlet : *valve*
    vent : *valve*
init $Plant_0 \triangleq mk\text{-}Plant(\text{CLOSED}, \text{CLOSED}, \text{OPEN})$

The fail-safe condition is represented in $Plant_0$ this may also be written as:-

$inlet(Plant_0) = \text{CLOSED} \wedge$
$outlet(Plant_0) = \text{CLOSED} \wedge$
$vent(Plant_0) = \text{OPEN}$

The plant controller may be specified as a single operation, thus:-

Plant_Controller()

ext wr *inlet* : *valve*
    wr *outlet* : *valve*
    wr *vent* : *valve*

pre $inlet(Plant) = \text{CLOSED} \wedge$
    $outlet(Plant) = \text{CLOSED} \wedge$
    $vent(Plant) = \text{OPEN}$

post $inlet(Plant) = \text{CLOSED} \wedge$

outlet(*Plant*) = CLOSED ∧

vent(*Plant*) = OPEN

This states that the operation has no input or output parameters. It has read and write access to all three state variables. Its pre and post-conditions are equivalent to $Plant_0$ i.e. the fail-safe condition.

5.5.2 Decomposition into a sequence.

In the example plant controller, assume for the moment that the controller just responds to one demand for service followed by one demand for shutdown. The operation of the controller can be specified as the sequential composition of two operations, called say *Service* and *Shutdown*. This is written:-

Plant_Controller: *Service*; *Shutdown*

Where:-

Service()

ext wr *inlet* : *valve*
wr *outlet* : *valve*
wr *vent* : *valve*

pre *inlet*(*Plant*) = CLOSED ∧
outlet(*Plant*) = CLOSED ∧
vent(*Plant*) = OPEN

post *inlet*(*Plant*) = OPEN ∧
outlet(*Plant*) = OPEN ∧
vent(*Plant*) = CLOSED

Shutdown()

ext wr *inlet* : *valve*
wr *outlet* : *valve*
wr *vent* : *valve*

pre *inlet*(*Plant*) = OPEN ∧

outlet(*Plant*) = OPEN ∧
vent(*Plant*) = CLOSED

post *inlet*(*Plant*) = CLOSED ∧
outlet(*Plant*) = CLOSED ∧
vent(*Plant*) = OPEN

The specification claims that the application of *Service* followed by the application of *Shutdown* is equivalent to the application of *Plant_controller*. This claim can be verified by the use of an inference rule for sequential composition. Using this rule, appendix C shows the proof of the validity of this design step.

This procedure can be generalised to longer sequences of operations. Therefore VDM proofs rules can be used to show that a sequence of operations will bring about a state transition which is identical to that of another single operation. Interestingly, unlike CSP and CCS, there is no mechanism in VDM to answer questions like "are two given sequences the same?". Having shown the simple case of sequential composition the next sections consider the more complex control constructs of selection and iteration.

5.5.3 Weakening specifications.

Examining the sequential composition rule above and other rules subsequently, it may appear that there should be certain equivalences between the pre- and post-conditions of operations and sub-operations. The apparent requirements are that:-

(a) the pre-condition of the first operation in a seqential composition should be equivalent to the pre-condition of the decomposed operation.
(b) the post-condition of the last operation in a sequential composition should be equivalent to the post-condition of the decomposed operation.
(c) the pre-condition of an operation should be equivalent to the post-condition of the operation preceeding it.

The following example serves to illustrate why these requirements are only partially true. Consider a decomposition of the operation *Service*. The decomposition will be a sequential composition of three operation each changing the state of one valve. The operations are defined as:-

Close_vent ()
ext wr *vent(Plant)* : *valve*
pre *vent(Plant)* = OPEN
post *vent(Plant)* = CLOSED

Open_inlet ()
ext wr *inlet(Plant)* : *valve*
pre *inlet(Plant)* = CLOSED
post *inlet(Plant)* = OPEN

Open_outlet ()
ext wr *outlet(Plant)* : *valve*
pre *outlet(Plant)* = CLOSED
post *outlet(Plant)* = OPEN

The decomposition is:-

Service : *Close_vent*; *Open_inlet*; *Open_outlet*

The proof that this design step is valid follows from the inference rule for weakening a specification and an extended form of the sequential composition rule. As shown in appendix C the application of these rules to the above decomposition leads to the conclusion that:-

{*pre-Service*} (*Close_vent*; *Open_inlet*; *Open_outlet*) {*post-Service*}

This is only possible by taking great care over which states are refered to in the various conditions. Remember that the operations in the decomposition should not and cannot be examined individually to determine whether they form a consistent decomposition, this does not make sense. It is the claim that the operations, taken together as a sequence of three operations, form a consistent decomposition which is to be proven or denied.

5.5.4 Decomposition into conditionals.

Most of the important points about decomposition have been illustrated in the above section dealing with sequential composition. In order to illustrate this consider an operation for which consults a sensor and then sets a flag according to the measurement. The state variables are:-

state *Detector* of

pressure : *sensor*

serve_flag : *flag*

end

sensor : **R**

flag = { SERVE, SHUTDOWN }

The operation can be defined as:-

Read_sensor ()

ext rd *pressure(Detector)* : *sensor*

wr *serve_flag(Detector)* : *flag*

pre true

post (*pressure(Detector)* $\leq$ 100 $\wedge$

serve_flag(Detector) = SERVE) $\vee$

(*pressure(Detector)* > 100 $\wedge$

serve_flag(Detector) = SHUTDOWN)

If this operation is to be decomposed into a conditional statement as follows:-

Read_sensor: if *pressure(Detector)* $\leq$ 100 then *Flag_serve*

else *Flag_shutdown*

where:

Flag_serve ()

ext rd *pressure(Detector)* : *sensor*

wr *serve_flag(Detector)* : *flag*

pre *pressure(Detector)* $\leq$ 100

post *serve_flag(Detector)* = SERVE

Flag_shutdown ()

ext rd *pressure(Detector)* : *sensor*

wr *serve_flag(Detector)* : *flag*

pre *pressure(Detector)* > 100

post *serve_flag(Detector)* = SHUTDOWN

then, as previously, there is a proof obligation to be satisfied and this is shown in appendix C. The conclusion drawn from this proof is that:-

{pre-Read_sensor}(if *pressure(Detector)* $\leq$ 100 then *Flag_serve*
else *Flag_shutdown*){*post-Read_sensor*}

and therefore the decomposition is consistent.

5.5.5 Decomposition into loops.

The final control construct to be considered is that of iteration. For consistent decomposition of an operation into a loop, the loop must be demonstrated to terminate as well as ultimately providing the desired state transition. Loop termination is easily demonstrated in loops where some variable is gradually increased or decreased and eventually meets some end point. However in many real-time systems loops often take the form below:-

```
WHILE NOT (pressure_alarm=ON) DO
        Control_pressure;
        Read_alarm
END
```

Intuitively this loop performs a clearly defined purpose: control the pressure while the alarm is not ON. Appendix C shows two examples of specifications for the decomposition. The first demonstrates how the proof rule for VDM works. The second shows a more difficult problem. In the second example there are two conditions to consider:-

(a) Firstly one in which the value of the pressure never rises above 2000. In this case the pressure alarm is always OFF and the loop never terminates.

(b) Secondly one in which the pressure rises above 2000. In this case the pressure alarm is set to ON. The loop test is therefore false and the loop terminates. The condition of the state on termination is equivalent to that specified by *post-Controller*.

This leads to the conclusion that the loop does not necessarily terminate, but it does under some conditions. Under the terminating conditions the state is in the condition specified.

The notion of triples of the form:-

$$\{pre\}\ S\ \{post\}$$

relies upon the termination of the operation *S*. This forces the conclusion that for certain decompositions into loops there is no formal proof mechanism in VDM which can show the consistency of such decompositions. The loop construct can be analysed to see if under certain circumstances it does terminate. The condition of the state upon termination can also be examined to show that it is consistent with the higher level specification. The issue of non-termination of loops is important for real-time specifiers. Using VDM it is possible to show whether termination of a loop is guaranteed or not. If termination is not guaranteed it can be shown under what circumstances termination is possible. Such analysis can provide valuable information about possible problems in a proposed design.

5.5.6 Data reification.

In the decomposition examples thus far, the data types used have been sets with enumerated membership. The implementation of such data types in a language like Modula-2 or Pascal is trivial. However the earlier example of a buffer was defined in terms of a sequence. This type of data structure has no direct implementation in these languages. The design of an implementation of the *Buffer* specification must therefore change the representation of the data type to some data type which exists in these languages. This process of changing from abstract mathematical data types to concrete programming data types is called data reification.

Once again, not surprisingly, proof obligations arise from the decision to use a particular concrete representation of an abstract data type. The first element of a proof of a data reification step is a retrieve function. This is a function which converts a concrete representation into the original abstract one. Formally, for an absract representation *Abs* and a concrete representation *Rep* the retrieve function has a signature:-

$$retr\text{-}Abs : Rep \rightarrow Abs$$

The most obvious property of such a retrieve function is that it should be adequate i.e. there is at least one representaion for each abstract value. Formally, adequacy is encapsulated in the proof obligation:-

$$\forall\ a \in Abs \cdot \exists r \in Rep \cdot retr\text{-}Abs(r) = a$$

With adequacy established, operations can be specified in terms of the new representation. There are two further proof obligations which arise from this design stage. These are to show that the new operations model the operations on the abstract data types. The first proof obligation is to show the correct modelling of the domain. It is stated as:-

$$\forall r \in Rep \cdot pre\text{-}A(retr\text{-}Abs(r)) = pre\text{-}R(r)$$

The second proof is to show correct modelling of the result of the operation application. It is stated as:-

$$\forall \underline{r}, r \in Rep \cdot pre\text{-}A(retr\text{-}Abs(\underline{r})) \wedge post\text{-}R(\underline{r}, r) \Rightarrow post\text{-}A(retr\text{-}A(\underline{r}), retr\text{-}A(r))$$

These proofs depend heavily upon properties of concrete data types being formally defined. Such definitions are not readily available although Bjorner [Bjorner82] does give an example for the Pascal language. Jones [Jones90] acknowledges that such proofs are probably best handled by informal constructive arguments.

5.6 Building and refining a specification - summary.

In this section the VDM notation for defining states and operations has been introduced. Also the concept of refining a specification for an abstract implicit definition towards a concrete representation has been introduced though operation decomposition and data reification. Operation decomposition allows a designer to define sequences of actions and control structures which perform specified tasks. The fulfilment of the tasks can be demonstrated by formal proofs. Data reification allows a designer to replace abstract mathematical data types with concrete representations in an implementation language. The adequacy of the representation can be demonstrated. Operations can be rewritten in terms of the new representation and the correctness of the new operations with respect to the old operations can be proven.

In theory it is possible for a designer to refine a specification into an implemetation in a particular language. Each step of this process can be formally verified and the correctness of the resultant program with respect to its specification can thus be shown.

6 ANIMATING FORMAL SPECIFICATIONS

6.1 Animating Formal Specification - an introduction.

An important part of the specification process is an attempt to define client requirements. As with any high quality specification there is a need for feedback from clients as to the accuracy of the formal specification in detailing their requirements. As the previous chapters have shown, formal specifications employ specialised mathematical constructs. Clients, as highly numerate engineers, may be able to grasp the broad concepts of a well-structured and presented formal specification. The dilemma for the software specifiers then is that clients may not comprehend all the of the formal specification. Familiarity with the notation used and the specification itself is required to reach a deeper understanding.

As mathematical notations appear to be a problem is it possible to remove them from discussions with the client? Hall [Hall90] suggests that a well written formal specification can indeed be rewritten without mathematics. The client is then presented with a more conventional natural language document. It is argued that because such a document is derived from a carefully constructed specification it is inherently clearer than a conventional text-based specification. This approach does have several advantages. The use of formal techniques to build a specification should mean that important issues within the system have been addressed. The attempt to describe something in mathematical trerms currently requires much greater thought than writing a text-based specification. The problem is that such descriptions may not address what the client sees or knows to be important issues as the version of the specification that the client sees is still a natural language document. The problems associated with such specifications were highlighted in chapter 2. Lastly and perhaps more importantly there are now two translations in this process, from informal requirements to formal specifications and thence to natural language document. This raises many questions, such as:-

* How is the translation from the formal specification to natural language to be accomplished?

* Who will verify that all the properties in the formal specification have been accurately represented in the new document?

* Will the resultant document be any easier for the client to analyse than other traditional types of documents?

A different approach to making the meaning of formal specifications accessible to non-specialists is that of animation. Animating formal specifications is not a single clearly defined concept. As with prototyping of software, there are a variety of very different approaches to animation. Broadly there are three types of animation, distinguished by the way in which information is presented to the client. The approaches are:-

(a) Text-based animations [Goguen82, Henderson86, McAsey85]. This is what is most commonly meant by animation. In this approach the formal specification is translated into an executable program. The specification is then exercised to determine its reaction to inputs. The output from such animations is text i.e. a listing of the current state or descriptions of the output.

(b) Diagram animations [Toetenel90]. At the center of this approach are diagrams, such as data flow digrams, which show the structure and flow of data between processes within a specified system.

(c) Graphical animations [Hekmatpour88]. The emphasis of this approach is to use pictures to mimic the appearance of the specified system. The properties of the formal specification can then be demonstrated in system-specific terms, e.g. menus and heirarchies of menus which behave in accordance with the specification.

A further distinction between animation techniques can be made along the lines of how the specification is executed. The two main approaches are:-

(a) actual execution. The specification is translated into a suitable language. The resulting program is then run with specific input values. The output variables of the specification would then be bound to the values defined by the specification.

(b) symbolic execution. The specification is translated again, but is run on symbolic input values. These may be either ranges of values, e.g. all valid inputs, or descriptions of values, e.g. x. The output from such execution would be either a range of values related to the input range or a literal description of the result. For example giving the input "x" to a function called square might result in the output of the literal "square(x)". Symbolic execution is commonly used with algebraic specifications as axioms used as rewrite rules provides a convenient model of computation.

Few of the specification languages which have been suggested for animation work are inherently

executable. So in order to execute a specification a translation to a programming language is necessary. Even for small specifications this translation is an error prone process. Problems arise in two main classes; firstly, the misunderstanding of the specification by the translator; secondly, bad translation of the specification, where the properties are well understood but the eventual implementation does not capture all the properties. Now it is very important to ensure that an animation is actually a true representation of the formal specification. The best solution to this problem is to perform the translation to an animation of a formal specification automatically. Such automatic translation cannot eliminate all errors, but it can ensure a consistent interpretation of the formal specification constructs and a consistent translation mechanism.

6.2 Examples of animation systems.

The following examples serve to illustrate the wide variety of approaches to animating formal specifications. They also give a helpful insight into the possible translation strategies for producing an executable model of a formal specification.

(a) Algebraic specification languages

(i) OBJ [Goguen82, Goguen84] is an executable algebraic specification language. The ObjEx toolset [Gerrard90] allows a subset of OBJ to be animated. Animations are based on term rewriting (see chapter 4) and are text-based.

(ii) me too [Henderson86]. Me too is an executable specification language related to the functional programming language Miranda. The language is embedded in a Lisp environment. Once again the interactions are text-based. There is no translation mechanism as functions defined by the specification language can be written as Lisp functions.

(iii) CSP, me too and Ada [Clarke90]. This system combines CSP and me too. Objects are specified using me too and communication processes between objects using a subset of CSP. The execution uses user instantiated versions of the defined objects. The progress of execution is controlled by the user who provides extra information about the timing of events. The progress of the execution is displayed as text which shows what events will happen next.

(iv) Algebraic specifications, LISP, Prolog and C [Antoy90]. The author's have defined a small algebraic specification language related to OBJ and Larch. Specifications are translated from this language to the target programming language by a tool. The system of term rewriting is used for execution and all input and output is text-based.

(v) MAL and Prolog [Booth87]. This system uses a subset of the FOREST formal notation MAL. Specifications are automatically translated to Prolog for symbolic execution. When executing the specification, the user chooses when events occur and may browse through the Prolog database to examine the state of the system. It is proposed that this work be continued to provide support for graphical animation of specifications [Atkinson91].

(b) Model-oriented specification languages.

(i) VDM and Prolog. The process of automatic translation of VDM to Prolog has been clearly defined [McAsey85]. Prototype Prolog programs produced by this process may be queried in the usual Prolog style. Interactions are thus question and answer sessions about the application of relations defined in the specification to user-supplied values.

(ii) VDM and Miranda [O'Neill89]. Another approach to animating VDM specifications is purpose-build miranda prototypes. The animations are text-based with the execution working with actual data values. The animation sessions have much in common with the use of a query language.

(iii) VDM to C [Hekmatpour88]. Translating VDM specifications into executable C programs has also been automated. In this system great emphasis is placed on the specification of user interface with support provided by the tool for adding menus and windows to a specification. When a specification is executed the user sees a representation of what the interface is specified to look like. The main focus of the system is data processing applications.

(iv) Z to Prolog [Dick89]. A scheme for transforming Z specifications to Prolog programs has been defined. Execution of the specification can produce all outputs generated from an initial state. It is possible to link the value of state variables to graphical entities to enhance presentation of the animation.

(v) VDM and Structured Analysis [Toetenel90]. In this approach to specification and animation, specifications are built in a graphical form with networks of interconnected objects. From this network a VDM specification is automatically derived. Using a further tool the VDM specification is animated. The progress of the animation are shown graphically as changes in the network.

(vi) Ina Jo [Kemmerer85]. Ina Jo has tools for both symbolic and actual execution. The merits and drawbacks of each appraoch are clearly illustrated by this system. The interactions are

all text-based even though it is aim at eliciting user requirements.

(c) Executable specification languages

(i) PAISley [Zave91]. The PAISley approach to executable specifications is an important piece of research in this field. Although not strictly a formal system PAISley does support formal reasoning about specification properties. The PAISley system consists of a number of tools for building and managing specifications. Through an interpreter tool PAISley specifications can be executed. Executions are text-based and the possibility of adding user-friendly interfaces to demonstrations is recognised.

(ii) RSLogic [Pacini87]. RSLogic is a diagrammatic method of capturing event histories. Properties of these diagrams can be reasoned about using a new formalism called RSLogic. Execution of specifications is symbolic and achieved by automated translation to Prolog. When executed the specification can be queried about possible histories leading to events. Answers to queries are in the form of listings of possible histories.

(iii) RSF [Degl'Innocenti90]. RSF is an event-based executable specification language. The specific application is timing constraints in real-time systems. Its approach to animation is almost identical to that of RSLogic.

(iv) TRIO [Ghezzi90]. The TRIO formal specification language is aimed at the specification of real-time systems. It is closely related to RSF and RSLogic in its approach to animation. Prolog is the implementation language and the execution of specifications can generate or test histories of events with respect to the specification.

The above work provides a guide to the possible approaches to animation and important issues that need to be addressed.

6.3 Important Properties of Animation Techniques.

The above pieces of research illustrate the important properties which need to be considered in building and animating formal specifications. The following major points illustrate the influence of the works on this thesis:-

(a) Structuring specifications. Specifications contain a great deal of information. In order to make specifications easier to deal with it is necessary to present that information in a orderly

and accessible way. The use of formal mathematical notations gives a degree of consistency to the presentation. However for large specifications there is a need for further ordering and structuring. Diagrammatic techniques to illustrate hierarchies and structures are used in a number of software engineering techniques such as Ward/Mellor and Jackson Systems Development. A technique for structuring VDM specifications has been developed. This technique is discussed later in this chapter.

(b) Animation style. The purpose of animation is to demonstrate important properties of the formal specification to those unfamiliar with formal notations. This enables observations to be made as to the correctness of the formal specification in describing the problem. Animations must therefore not rely on specialised notations comprehensible only to software engineers. From the work carried out with animated prototypes of real-time systems it is clear that computer-animated pictures are a useful way of presenting such properties.

(c) Formal notation. The automated animation of the complete set of constructs in most formal notations is not possible. This is because:-

(i) Data structures may be defined which are infinite or near infinite. To view even a single instance of such structures might take many years. Such structures can, however, be defined mathematically and then viewed as sets of related instances.

(ii) Functions may be defined with infinite or near infinite domains and ranges. Once again to view possible values within these sets would take a very long time.

(iii) Functions may be defined which have only a single solution in a very large set of possible solutions. In a generalized solution this would lead to a lengthy search of the possible solutions.

(iv) Functions may be defined as non-deterministic. Such functions have more than one possible solution for their final states for a given initial state.

It is thus necessary to restrict the constructs used. It is important however to ensure that any restrictions made do not preclude the use of the deductive apparatus of the formal system.

6.4 Animating Real-Time Embedded Systems - A Conceptual and Theoretical Framework.

6.4.1 Introduction.

The key aspects of animation prototyping of formal specifications for real-time systems are:-

* expressing the specification in VDM.
* using a diagram to express the structure of the specification.
* using a structuring methodology based on hierarchical decomposition principles.
* supporting the structuring methodology by an appropriate diagramming method.
* restricting the VDM subset for simplicity.
* ensuring that the mathematical requirements are faithfully fulfilled (maintained) when decomposition takes place.
* translating the specification into executable code which runs an animation on a computer.

6.4.2 Specifying systems using VDM.

The use of formal methods gives a firm foundation for the development of the system. As shown in chapter 5, VDM provides a mechanism for expressing a system's functional specification. The formal specification can underpin the subsequent software development process as VDM supports a very rigorous approach to software development. Furthermore, with its mathematical properties precisely stated, a VDM specification can form the basis for the automated production of a prototype.

6.4.3 Using structure diagrams with specifications.

Formal specifications of real-time systems are large and complex mathematical texts. The structure and ordering of functions within the specification needs to be highlighted. The use of diagrams to represent specifications is widespread in informal structured design techniques such as YSM, JSD and others. The use of a diagramming method to complement the mathematical rigour of the formal specification will increase the accessibility of the information and hence aid the specifiers.

6.4.4 A structuring methodology for specifications.

In real-time systems the ordering of operations and their effects on the system is of prime importance. The role of the VDM specification is twofold. First, it defines a model for the state of the system. Second, it defines a set of operations which may change affect that state. VDM operations define state transitions. An initial state, or set of states, is defined and the relationship between those states and the final state, or set of states, is also defined. Such state transitions can be decomposed to give more information about system behaviour. The constructs used in the decomposition are sequence, selection and iteration. The mathematical foundations of these decompositions are shown in chapter 5. Thus a single defined operation can be decomposed into a series of consecutive transitions which begin and end in the precisely defined states of the parent operation.

6.4.5 Diagrams for structuring.

The particular diagramming method chosen reflects the style of the decomposition. The Jackson Structured Programming approach to structured diagramming supports the decomposition constructs proposed here. A JSP diagram clearly shows the ordering relationships between parent operations and their children. A commercially tool, JSP-TOOL, is available which facilitates the drawing, storing and retrieving of JSP diagrams. Furthermore, JSP-TOOL supports the generation of text from diagrams which makes automatic prototype production feasible.

6.4.6 A subset of VDM

As discussed earlier, it is not practical to animate specifications written using the complete VDM notation. Consequently, bearing in mind that real-time systems are to be specified, a subset of the VDM notation has been defined. This subset is discussed in detail below and its application to practical systems is shown in chapter 8.

6.4.7 Maintaining consistency in decompositions.

The emphasis of VDM design is the preservation of specification properties throughout the design process. The deductive apparatus of VDM allows such preservation to be formally verified. The consistency of a decomposition is shown in two ways. Firstly, the interfaces between operations in the decomposition are checked. This involves the checking of pre and post-conditions of consecutive operations and the careful tracing of state variable changes brought about by those operations. Secondly, the operations and the defined decomposition structure are checked to ensure that they bring about the state transitions defined by the decomposed operation. The subset of VDM has been defined so that this verification is possible within the deductive framework of VDM.

6.4.8 Automatic prototype production.

The majority of people invloved in the production of software-based systems are not specialists in formal methods. Yet verification of the properties of a formal specification by those people is nonetheless essential. Animation prototyping is one approach to making key properties of the formal specification demonstrable to non-specialists. Automatic translation into executable code which runs an animation on a computer is the key to preserving the meaning of the specification. A practical process for producing such animations from a formal specification is presented in chapter 7.

6.5 Definition of a Subset of the VDM Notation For Use in Real-Time Embedded Systems.

A complete definition of the syntax of the notation used in this project is contained in Appendix E. The aim of this section is to give an explanation the notation and the reasons behind the choice of the subset.

The complete VDM specification language is not inherently suitable for animation. Therefore, to create specifications which can be animated, it is necessary to choose a subset of the language. The most important consideration is that the notation and its animation are to be used at a very early stage in the software development process. Therefore, there is a need to use an abstracted view of the system, as too much detail will inhibit discussions and obscure important requirements. Also any notations used must describe the software in a clear and understandable manner. It is proposed that the following approach is suitable for these purposes.

Specifications are to be built in a model-oriented style. A model-oriented specification comprises:-

(a) A definition of data types. This is limited to enumerated types plus one composite data type. This composite data type is used to represent the system state.

(b) A definition of a state.

(c) A description of the set of the possible initial states.

(d) A list of operation definitions.

This approach was chosen in preference to an algebraic, functional style as it was felt that specification of this type were easier to comprehend. The writing, manipulation and checking of model-oriented specification is well-suited to the heirarchical decomposition of real-time embeded systems.

At a practical level, VDM makes extensive use of special fonts in its syntax. Writing VDM specifications in this style on a computer requires a special editor or a text post-processor (such as Latex). The alternative adopted here is to use only a single font and make more use of extra syntactic symbols. The use of such symbols does not change the semantics of the statements it merely makes the distinctions between the components in a different manner. The following definitions thus make use of only ASCII symbols, available in almost all editors.

6.6 Definition of the Set of States and Data Types.

6.6.1 Data types.

In order to define the state of a system it is necessary to define data types. In this project only very simple data types are used. The use of very simple data types is seen as necessary to limit the task of translation to a manageable size. The important issues of assessing the correctness of specifications are still highlighted with these data types. The constructs allowed in the subset are chosen carefully so that the type of problems being considered can still be expressed. The use of complex, dynamic data structures is not as important in embedded real-time applications as in other types of software.

The types which may be defined are sets of elementary values, enumerated explicitly, e.g.

```
Valve = { OPEN, CLOSED }
Alarm = { SET, RESET, OVERRIDE }
```

A true Boolean type is not part of the SIMSCRIPT language. This lack of this highly useful type is seen as a major drawback of using SIMSCRIPT for further development work.

A single composite data type is allowed. A composite data type is one composed of a number of fields; each such field has a type as above and a value, e.g.

```
compose PlantState of
        InputValve : Valve
        OutputValve : Valve
        PressureAlarm : Alarm
end
```

6.6.2 State definition.

The state name and type are defined in the following manner:

```
state
        PState : PlantState
```

The set of initial states is defined by a logical expression which evaluates to true for all valid initial states e.g.

```
initial
        (       InputValve(PState) = OPEN
         and    OutputValve(PState) = OPEN
         and    PressureAlarm(PState) = RESET )
or      (       InputValve(PState) = CLOSED
         and    OutputValve(PState) = CLOSED
         and    PressureAlarm(PState) = SET )
end
```

6.7 Definition of Operations

6.7.1 The basic elements of an operation specification.

The definition of operations is done implicitly as suggested by Jones book [Jones90]. There are four main parts in an implicit operation description of this type:-

* The operation signature;
* The external clause;
* The pre-condition clause;
* The post-condition clause.

These are described below.

6.7.2 Operation signature.

Each operation has a unique name. Overloading of names is not allowed as this may give rise to confusion. This name is followed by a list of input arguments with types. This list may be empty. After this comes a list of output arguments with type. Again this list may be empty. For example:-

```
And_Gate ( Input1 : Logic ; Input2 : Logic ) Output : Logic
```

6.7.3 External clause.

The external clause of an operation details the fields of the state variable that the operation affects and the type of effects caused. The simplest effect is to read a field. This means that an operation uses

the value of the particular field but does not change it. This would be specified as follows:

rd PressureAlarm(PState) : Alarm

The second effect is a read/write action on a field. This means that an operation uses the value of the field as it was when the operation began and also the operation may change the value of that field. This would be specified as follows:

wr InputValve(PState) : Valve

6.7.4 Pre-condition clause.

The pre-condition clause defines the range of values over which the operation effects are defined. The pre-condition is a truth-valued function. E.g.

pre true

This means that the operation is defined for all possible input values and states. Alternatively:

pre InputValve(PState) = CLOSED
and PressureAlarm(PState) = SET

This means that the operation is only defined when this expression evaluates to true i.e. the valve is open and the pressure alarm is set. For other values the operations outcome is undefined.

6.7.5 Post-Condition Clause.

The post-condition clause defines a logical relationship between values of the input, output and external variables for the operation. This relationship, in the form of a truth-valued function, evaluates to true in all cases where the pre-condition evaluates to true. If the pre-condition evaluates false then the post-condition may evaluate to true or false i.e. the outcome of the operation is undefined.

In very simple cases the post-condition just specifies the relationship between input and output arguments. However with external variables which are changed by an operation, there is a need to differentiate between the initial and final values of these variables. Jones suggests the use of a hook i.e.

$\vec{x}$ for the initial value.

x for the final value;

There is no ASCII representation for this notation. Therefore the use of the "Z" [Spivey89] notation of a prime (') to indicate initial values has been chosen here, i.e.

x' for the initial value;

x for the final value.

It is further defined that all non-determinism should be explicitly stated. Non-determinism is when there are many possible final states for a give initial state. Avoiding implicit non-determinism means that all variables which are changed by the operation must appear in the post-condition. Furthermore, all such variables must have their possible final values explicitly defined. Implicit non-determinism often arises when disjuncts occur in the post-condition.

A typical post-condition, shown in the context of its operation, would be:

```
Valve_Controller ( )
ext     rd PressureAlarm(PState) : Alarm
        wr InputValve(PState) : Valve
pre true
post            (     InputValve(PState)' = CLOSED
                and   PressureAlarm(PState) = SET
                and   InputValve(PState) = OPEN )
        or      (     InputValve(PState)' = InputValve(PState)
                and   PressureAlarm(PState) = RESET )
endpost
```

This expression has two mutually exclusive outcomes, dependent on the initial condition of the valve and pressure alarm:-

* the input valve is opened if the pressure alarm is set
* the input valve state does not change if the pressure alarm is reset.

In this operation the final state of the InputValve is uniquely defined by the state of PressureAlarm.

A simple operation which has explicit non-determinism in it may be defined as follows:-

```
Sensor = { HIGH, NORMAL, LOW }

Sensor_Reader ( )
ext     wr PressureSensor(PState) : Sensor
pre true
post            PressureSensor(PState) = HIGH
        or      PressureSensor(PState) = NORMAL
        or      PressureSensor(PState) = LOW
endpost
```

In this operation the final value of the pressure sensor is completely unrestrained by any initial conditions. It may take on any value within its type.

As post-conditions become more complex, with nested disjuncts and conjuncts, ensuring that no implicit non-determinism has been specified becomes increasingly difficult. The use of mutually exclusive disjuncts is essential to eliminating implicit non-determinism. This is an important problem for specifiers of safety-critical real-time systems. If the final values of variables are not explicitly constrained then a valid implementation is one in which these variables take on expected values. Such problems can only be completely eliminated by proving that disjuncts within the post-condition are mutual exclusive.

A simple operation which shows implicit non-determinism for the value of OutputValve(PState) is:-

```
Valve_Controller2 ( )
ext     rd PressureAlarm(PState) : Alarm
        wr InputValve(PState) : Valve
        wr OutputValve(PState) : Valve
pre true
post            (       PressureAlarm(PState) = SET
                and     InputValve(PState) = OPEN )
        or      (       PressureAlarm(PState) = RESET
                and     InputValve(PState) = CLOSED
                and     OutputValve(PState) = CLOSED
endpost
```

In this operation, in the case where the pressure alarm is set, the final state of the output valve is unconstrained. An implementation developed from this specification could behave in three different ways, it could:-

* always close the output valve.
* always open the output valve.
* either open or close the output valve randomly.

Either one of these implementations would be provably correct with respect to the specification. This is clearly not a desirable feature of a specification for a safety-critical system. As this is a general property of any non-deterministic specification, specifiers must be very careful about writing such specifications. This is the reason for insisting that all non-determinism is made explicit.

6.8 Definition of the Ordering of Operations.

6.8.1 Decomposing operations.

In real-time systems the ordering of operations is important. The ordering is introduced into the specification as the operations are decomposed. This decomposition is represented using structure diagrams are used as part of the formal specification. Partly this is done to help in the organisation and understanding of the specification. Mainly though, as timing information not covered explicitly in the specification, the structure of the specification does however deal with the ordering of events. The specification language deals mainly with changes in discrete variables. It is therefore not suitable for prototyping numerical algorithms. The main aim of this technique is to accurately specify the operation of real-time embedded systems. The state of the system and changes in that state are the main issues to be resolved.

Operations are decomposed using standard procedural language constructs i.e. sequence, selection and iteration. The specification is formed in a hierarchical, top-down fashion as in Figure 13. Consider the following example:-

> A system has an operation A which is to be decomposed into a sequence of two operations B and C in that order. Operation A has two conditions associated with it pre-A and post-A. Similarly B has pre-B and post-B, while C has pre-C and post-C. Stated informally, in order to show that using the sequence B followed by C has the same effect as using A the following

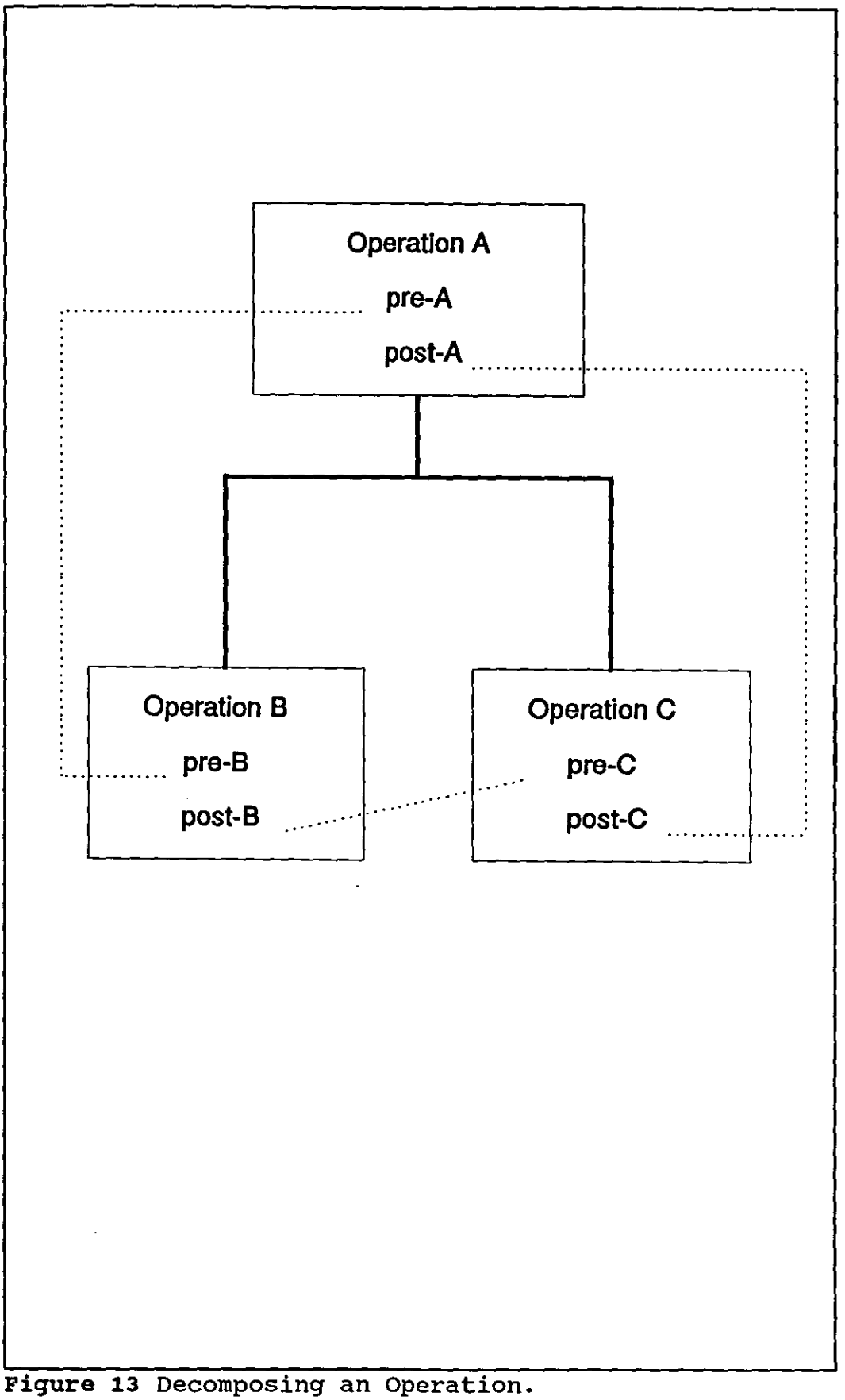


Figure 13 Decomposing an Operation.

conditions must be met:-

(i) all sets of values which satisfy pre-A must satisfy pre-B: i.e. B must be defined over the same (or wider) range as A. As described in chapter 5, the concepts of weakening a specification apply to the decomposition. For example, the higher level requirement to double a positive number can be satisfied by an operation which will yield double any number (positive or negative).

(ii) for all the values in (i) the resulting sets of values which satisfy post-B must also satisfy pre-C. That is C must be defined over the same (or wider) range of values as the range of results produced by B operating on the range of values defined for A.

(iii) for all the values in (ii) the resulting sets of values which satisfy post-C must also satisfy post-A. i.e. the sequence B followed by C must produce results which satisfy the relationships defined by A.

A formal demonstration of the correctness of the decompositions is described in chapter 5. Similar lines of reasoning can be applied to decompositions using condition and selection constructs. This approach provides a way for formally checking the decomposition of a specification.

The decomposition can also be demonstrated to be correct informally through the use of animation. The animation of operations and sequences of operations is seen as an important aid to effective communication with the client. The structure of the VDM animation tool is described in chapter 7.

6.8.2 Representing decomposition with diagrams.

Ideally when building diagrams to describe operation decomoposition one boxes would be used for each operation. The structure of the diagram then represents the decomposition of the operations. Different boxes can be used to describe the different decomposition structures; sequence; iteration and selection as in Figure 14. The VDM text describing each operation is stored in a file which is associated with the relevant box on the diagram as shown in Figure 15. Information regarding the state and initial conditions can be stored in another text file.

For practical reasons (described in chapter 7) the following approach has been adopted:-

(a) Type definitions, state definitions and initial states are stores in a "header" file. This is a file associated with the whole diagram.

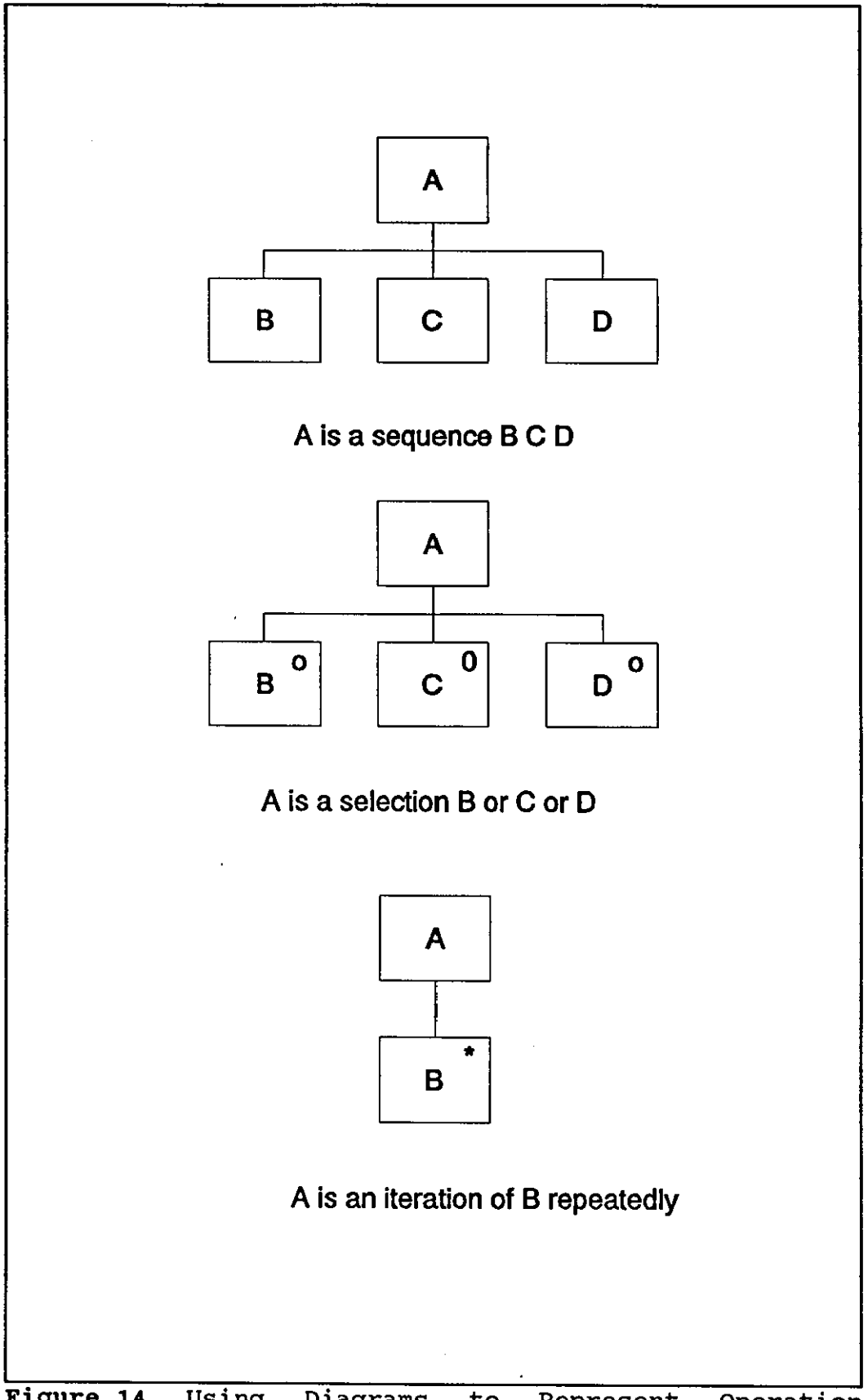


Figure 14 Using Diagrams to Represent Operation Decomposition.

(b) If a box on the diagram represents a single operation which is not decomposed then an "operation box" is placed below this box as in Figure 16. Operation boxes have text associated with them. The text associated with this box is the VDM specification of the operation.

(c) If a box represents a single operation which is to be decomposed into a combination of other operations then an operation box is placed at the left hand end of the decomposition sequence as in Figure 17. The text associated with this box is the VDM specification of the WHOLE operation i.e. what the composed sequence is supposed to do.

The full definition of the decomposition style is given in the EBNF description in Appendix E.

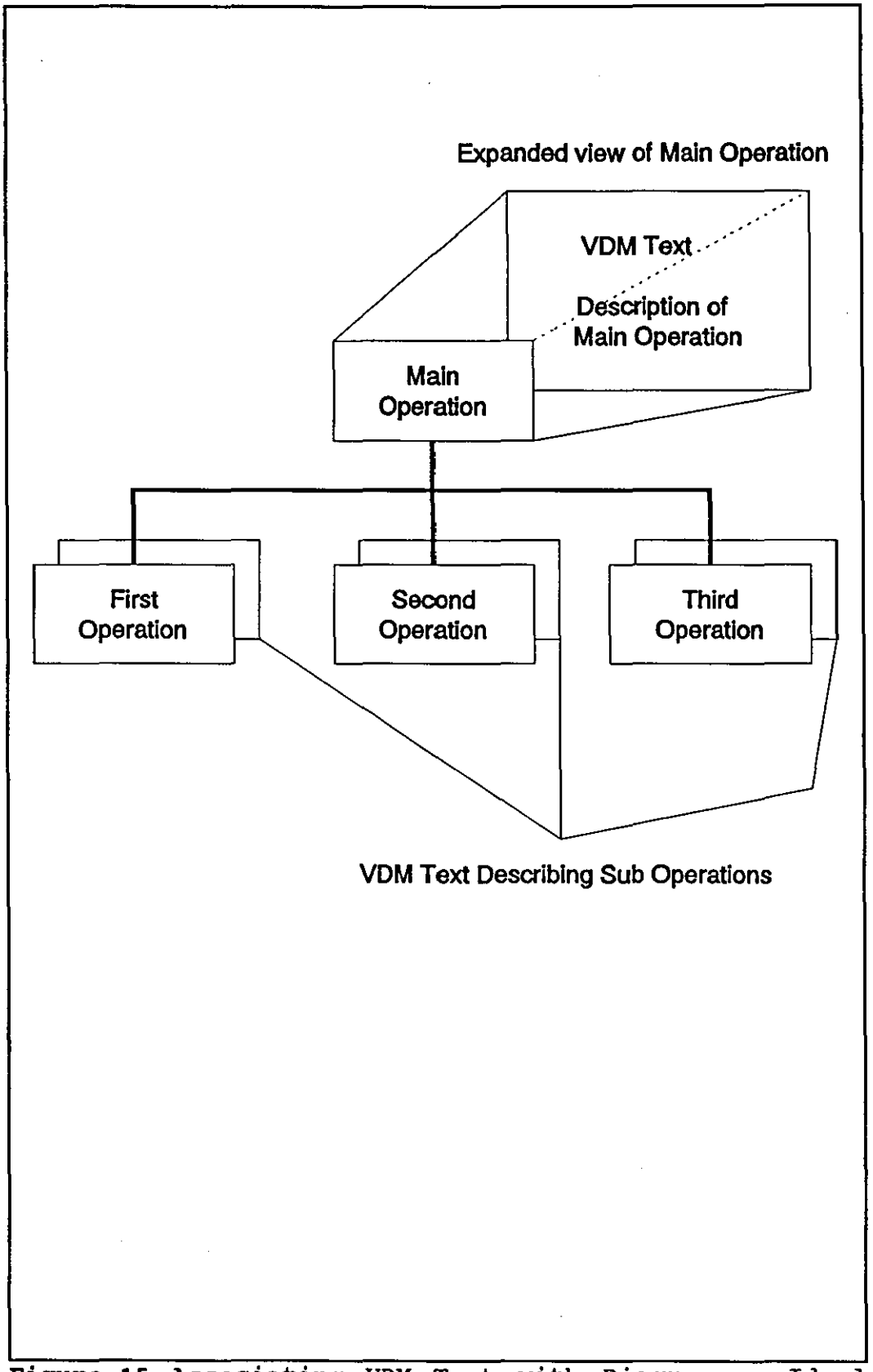
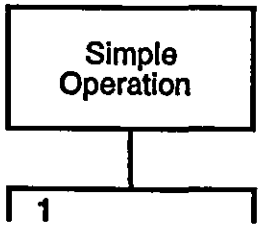


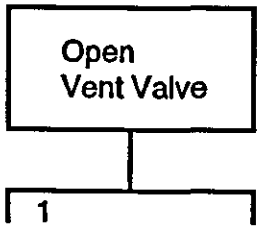
Figure 15 Associating VDM Text with Diagrams - Ideal Method.



Operation List

1 VDM Text Describing Simple Operation

(a) Basic layout



Operation List

1 OpenVentValve ( )

wr VentValve(PState) : Valve

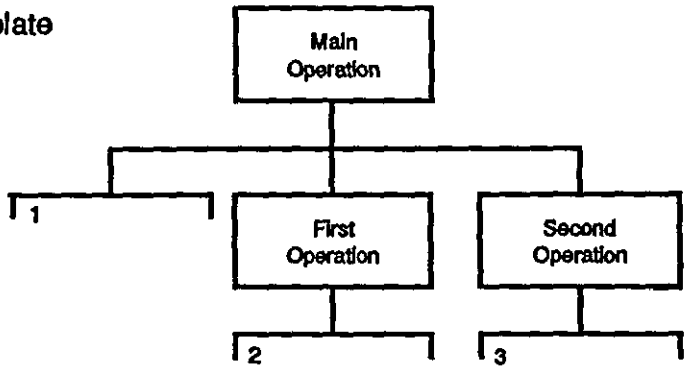
pre true

post VentValve(PState) = OPEN

(b) Specific example

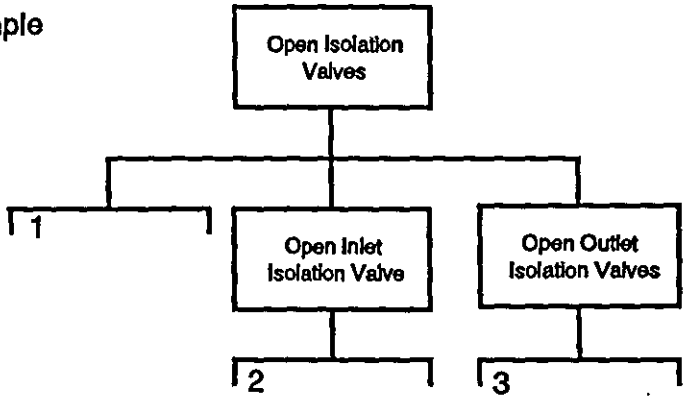
Figure 16 Associating VDM Text with a Single Operation.

(a) Template



Operation List
1 VDM text describing Main Operation
2 VDM text describing first operation
3 VDM text describing second operation

(b) Example



Operation List

```
1 OpenIsolationValves ( )
    wr InletIsolationValve(PState) : Valve
       OutletIsolationValve(PState) :Valve
    pre true
    post InletIsolationValve(PState) = OPEN
       OutletIsolationValve(PState) = OPEN

2 OpenInletIsolationValves ( )
    wr InletIsolationValve(PState) : Valve
    pre true
    post InletIsolationValve(PState) = OPEN

3 OpenOutletIsolationValves ( )
    wr OutletIsolationValve(PState) : Valve
    pre true
    post OutletIsolationValve(PState) = OPEN
```

Figure 17 Associating VDM Text with a Decomposed Operation.

7 THE ANIMATION PROCESS IMPLEMENTATION STRATEGY.

7.1 Overview of The Animation Process.

The purpose of the animation process is to produce an animated prototype which can demonstrate key properties of a formal specification to non-software specialists. The animation process begins with a VDM specification (written in the style described in the previous chapter) and ends with an graphically animated prototype.

The animated prototype consists of two parts:-

(a) A model of the system's behaviour. This is derived by translating the formal specification into executable code.

(b) Computer generated pictures which show the specified changes in the system's state.

The animated prototype is executed using the SIMSCRIPT II.5 package. The process of creating an animation is shown in Figure 18. The tools which are used to perform the different functions are shown in the figure. These are;

* JSP-TOOL for specification construction. JSP-TOOL is a structured diagramming tool. Building a diagrammatic tool such as the one described as "ideal" in chapter 6 was not possible within the timescale of the research project. It was thus necessary to find an already existing tool and adapt it to this use. JSP-TOOL provides the facilities to draw, store and retrieve diagrams according to the Jackson Structured Programming style. Text can be attached to these diagrams. Unfortunately, the style of diagrams in JSP-TOOL does not match exactly the ideal described in chapter 6 and hence the modified approach to diagramming described has been developed.

* JT-DOC for specification text generation. JT-DOC converts the JSP-TOOL structure diagram and its associate text into an ASCII text file.

* TP2 for text preprocessing. JT-DOC adds a great deal of extra information such as pagination and dates to the specification text. These unwanted characters are removed by the TP2 utility. TP2 is custom written in Modula-2.

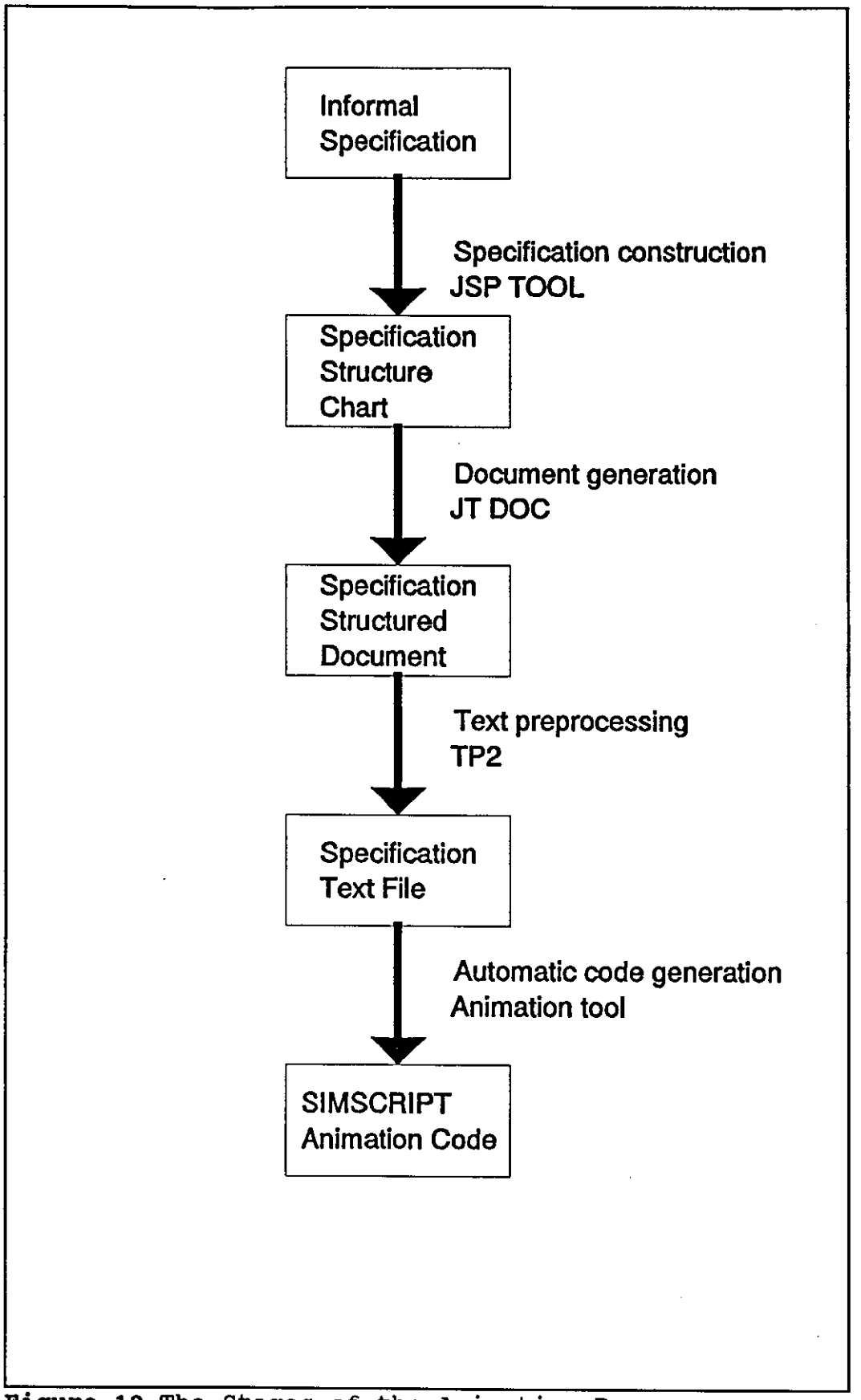


Figure 18 The Stages of the Animation Process.

* The automatic code generator. This custom made tool, written in SIMSCRIPT, performs the task of translating the specification into an executable SIMSCRIPT program.

Adding pictures and the routines to control the detailed aspects of user input is the final stage in the process. These routines are desrcibed later in this chapter.

The automatic code generator is the central theme of this work. It works in two phases; a parsing phase and a translation phase as shown in Figure 19. The task of the parsing phase is to take the text file containing the specification and analyses its structure according to the grammar tree specified. The details of the parser are described in the next section. The result of the parsing phase, in the form of a parse tree and a symbol table, is then used by the translation phase to produce a text file containing a SIMSCRIPT program. This SIMSCRIPT program can then has graphical output added to it. It is then compiled and executed.

The integration phase is the final stage of the animation process. The construction of the pictures used in screen displays is not automated. However, some support to reduce the effort involved in writing code to support graphical output has been produced. This support is described in the final section of this chapter.

7.2 Parser Design and Implementation.

7.2.1 The basic design of the parser.

The design of the parser is closely linked to the syntax of the specification language. The parser's task is to interpret the structure of the specification.

The approach chosen is to convert VDM's syntax to a one-track grammar [Bornat79]. A one-track, or one symbol look ahead, grammar is one in which the parser reads in one symbol at a time from the specification. The grammar is structured such that there is only one way to interpret this symbol. The parser may have many alternatives to search, but only one can be a correct interpretation of the structure. In order to achieve this conversion to a one-track grammar arbitrary symbols are added into the syntax of the VDM, e.g the symbols:-

"endpost" to mark the end of the post-condition expression;

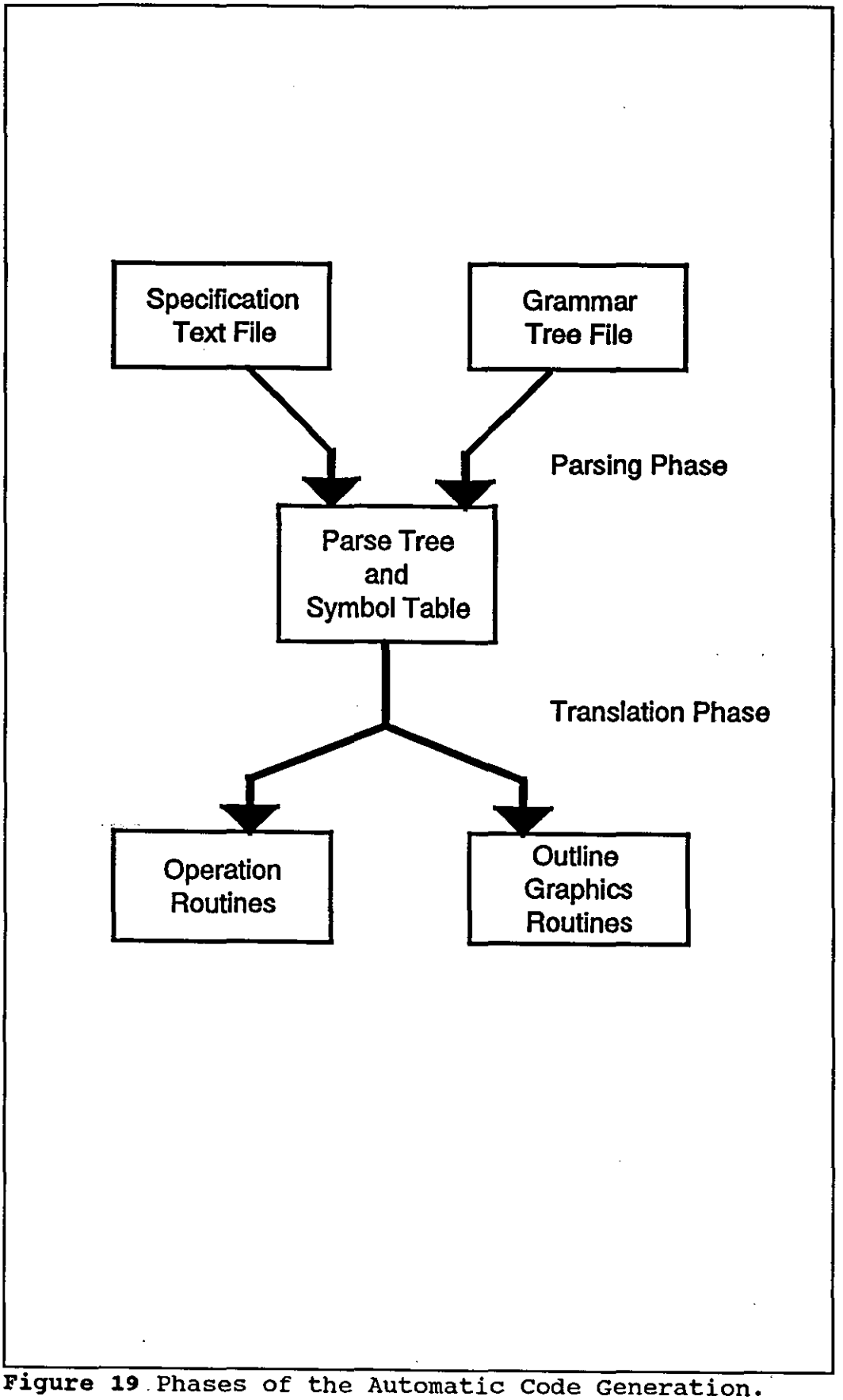


Figure 19 Phases of the Automatic Code Generation.

"endspec" to mark the end of the specification.

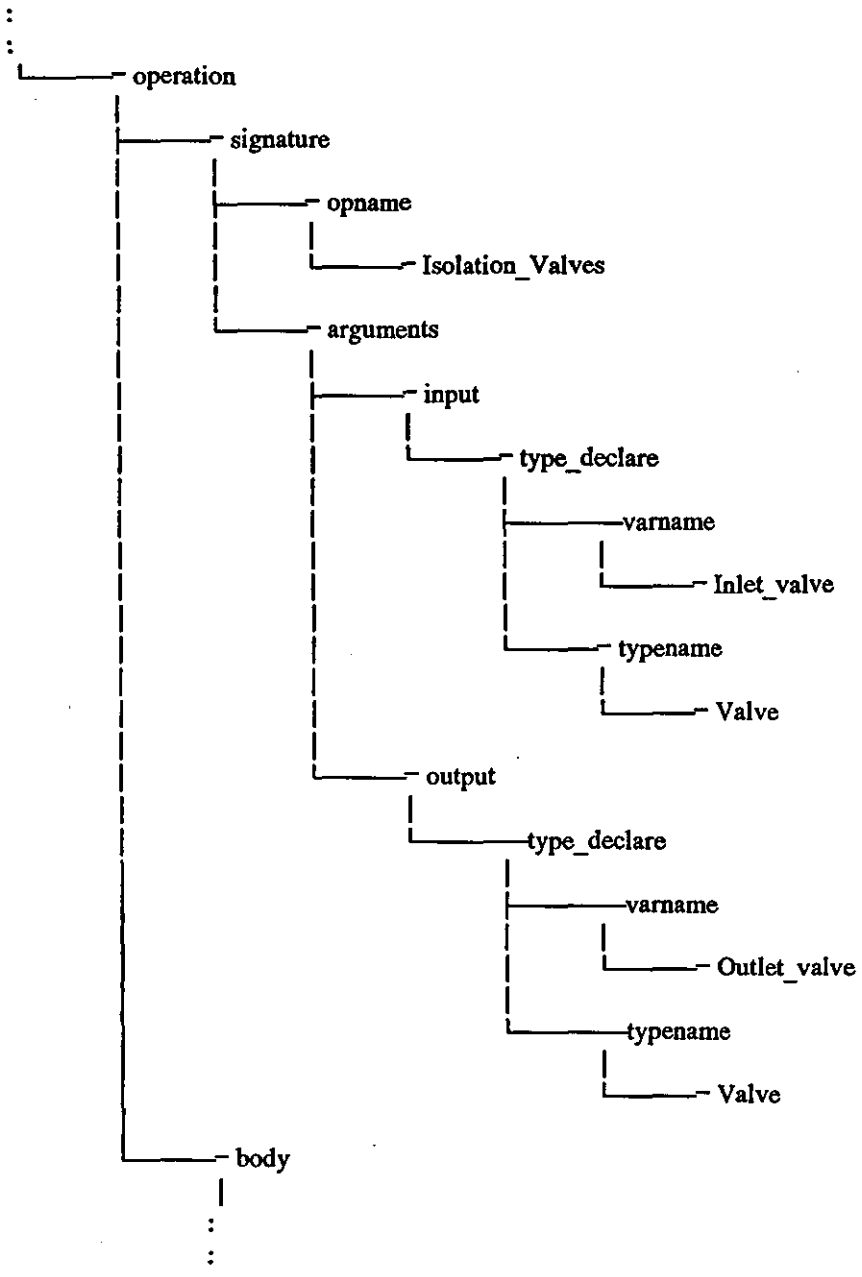
It is important to realise that the semantics of the language remains the same. Only the concrete realisation of that language is affected. The concrete realisation of VDM is a secondary issue; the semantics of the language is far more important.

The use of the one-track grammar analyser is augmented by an operator precedence analsyer to parse the many logic expressions found in a specification. The function of the operator precedence analyser is described below.

The final result of the parsing is a parse tree and a symbol table. The parse tree is a binary tree which defines the structure and meaning of the varuious parts of the specification. The nodes of the binary tree have types that define the particular structure that they represent and pointer to further nodes which represent the constituent parts of that structure, e.g. the parse tree which defines the signature of the operation defined as:-

```
Isolation_Valves ( Inlet_valve : Valve ) Outlet_valve : Valve
...
```

can be shown graphically as:-



The first node states that the section of the specification described by this part of the parse tree is an operation. The first sub-node (signature) represents the signature section of the specification. There are two sub-sections to a signature; an operation name (opname sub-node) and an arguments (arguments sub-node) list. The arguments sub-section has two further sub-sections; an input list (input sub-node) and an output list (output). This structure continues until the whole specification has had all its various sections represented by the appropriate pattern of nodes and sub-nodes.

7.2.2 Grammar trees and parsing.

The parser is structured to interpret any given one-track grammar. The grammar used to interpret a specification is defined in a grammar tree. In this way different grammars can be tried out or existing grammars extended simply by modifying the grammar tree. The parser implementation can interpret any valid text file according to any valid grammar tree. A grammar tree is binary tree built using four types of node; terminal, non-terminal, alternative and action nodes. Each node has a special role in the parsing:-

(a) Terminal nodes. These nodes, denoted by a 't', contain a terminal symbol and a pointer to the next node in the tree. The terminal symbol is a string of characters. If the current symbol being examined matches the terminal symbol then the path used to traverse the grammar tree defines the structure of the input to this point. If it does not match then the parser must backtrack through the graph to find an alternative interpretation of the input symbol.

(b) Non-terminal nodes. These nodes, denoted by an 'n', contain two pointers to further nodes in the grammar tree. These nodes act like procedure calls, in that part of the graph may be used repeatedly to interpret similar input structures. The first pointer points to the part of the tree which may be used to interpret the present input symbol. The second pointer points to the next node in the tree.

(c) Alternative nodes. These nodes, denoted by an 'a', contain pointers to two alternative branches which can be searched in order to interpret the input structure.

(d) Action nodes. These nodes, denoted by 'action', are used to construct the parse tree. They contain a text string which denotes the type of node which is to be added to the parse tree. Alternatively they contain special instructions to the parser to use routines to analyse expressions. They also contain a pointer to the next node in the grammar tree.

The complete grammar tree for the VDM subset is given in appendix F.

7.2.3 An example of how the one-track grammar works.

In order to better understand how the parser works, consider the following simple example. The phrase to be parsed is an operation's pre clause. The text is:-

```
pre
        a = b or c = d
post ....
```

The relevant part of the grammar is :-

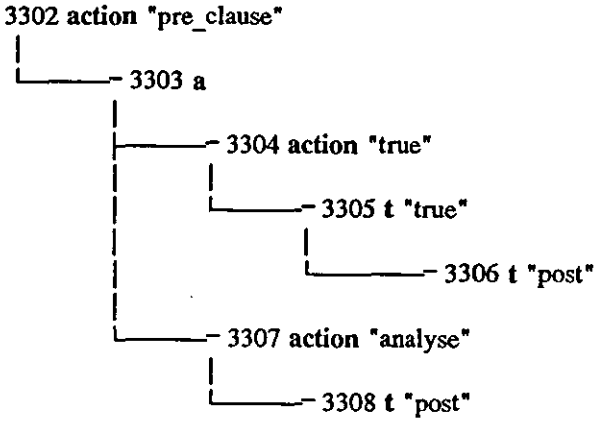
```
OpBody = ExternalList PreCondition PostCondition.
PreCondition = "pre" PreDeclare.
PreDeclare = "true" | Expression.
PostCondition = "post" PostExpression "endpost".
```

Briefly this states that an operation body (OpBody) consists of three consecutive parts; an external clause (ExternalList); a pre-condition clause (PreCondition) and a post-condition (PostCondition). The pre-condition consists of two parts; the word "pre" and the details of the pre-condition (PreDeclare). The details of the pre-condition consists of either the word "true" or an expression (Expression). Expressions and their interpretation are defined elsewhere in the grammar tree. The post-condition consists of three parts; the word "post", an expression (PostExpression) and the word "endpost".

The relavent section of the grammar is defined as a tree is :-

3302	action	3303	0	pre_clause	
3303	a		3304	3307	!
3304	action	3305	0	true	
3305	t		3305	3306	true
3306	t		3306	0	post
3307	action	3308	0	analyse	
3308	t		3308	0	post

Graphically this can be represented as:



The parser works as follows. Having recognised the item "pre" as the beginning of the pre clause the parser reads the next input item, which is "a", and is directed to node 3302. Node 3302 is an action

node. This causes the parser to add a node to the parse tree. The type of node is "pre_clause". The action node points to node 3303 as the next node. Node 3303 is an alternative node. The parser records this fact by adding a record of its current position in the grammar tree to a stack. At this stage the parse tree, stack and current node are as follows:-

parse tree	stack		current node
pre_clause	a	3303	3303 a 3304 3307 !

The parser then takes the first path which is node 3304. Another action node adds a "true" node to the parse tree and the parser moves on to node 3305. At this stage the parse tree, stack and current node are as follows:-

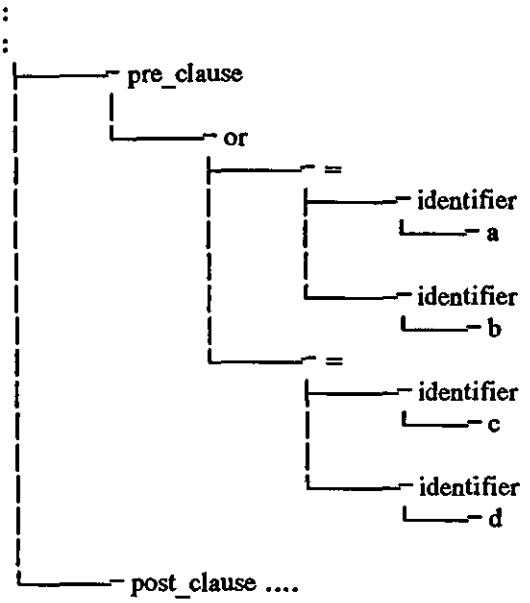
parse tree	stack		current node
pre_clause | └——— true	a	3303	3305 t 3306 0 "true"

Node 3305 is a terminal node. The parser compares the contents of this node ("true") and the current item ("a"). These items are not the same and therefore this path is not a correct interpretation of the input and the parser must backtrack across the grammar graph. To backtrack the parser returns to the last alternative (node 3303) and takes the second path to node 3307. At this stage the parse tree, stack and current node are as follows:-

parse tree	stack	current node
pre_clause		3307 action 3308 0 analyse

Node 3307 is a special action node which tells the parser to treat the following text as an expression to be analysed. This analysis is done using the operator precedence analysis routines. When this analysis has finished the current item will be "post" and the parser will be at node 3308. This is a terminal node and there is a match. There is no pointer to subsequent nodes from node 3308 and this indicates that the current phrase has been completely matched.

The partial parse tree for this phrase is:-



To interpret the rest of the specification, the parser then returns to the nearest non-terminal node. If this node contains a pointer to more nodes then the parser pursues that path. If not, the parser returns to and checks other non-terminal nodes.

7.2.4 Operator precedence analysis.

The purpose of the operator precendence analysis routines is to parse logical expressions. These expressions are of the form "A and B or C". The question to be resolved is should this be seen as;

"(A and B) or C"

or "A and (B or C)"?

Expressions are parsed as a left-hand side sub-expression and a right-hand side sub-expression connected by an operator. Each operator is assigned two priorities; one for right and one for left. The priorities used in the implementation of the parser are:-

Operator	Left	Right
or	1	2
and	1	2
=	3	2
(	6	0
)	0	6
identifier	3	4

The expression analyser begins by getting the first symbol. It then gets the operator. The current

operator's left priority is compared with the previous operator's right priority. If the current priority is greater then the parser must begin a new sub-expression. This it does by recursively calling itself and treating the returned sub-expression as the right hand side of the current sub-expression. However if the previous priority is greater then the current sub-expression is returned. Pairs of brackets can be used to enclose sub-expressions explicitly and so avoid any confusion about possible meaning. The choice of equal priorities for "and" and "or" in the evaluation enforces the use of brackets in this implementation.

7.3 Translation to SIMSCRIPT Source Code.

7.3.1 General translation strategy.

The translation phase of the animator takes the parse tree and the symbol table and from them produces SIMSCRIPT code. The code produced by the translation consists of :-

(a) A "PREAMBLE" containing variable declarations for the defined data types and associated declarations.

(b) A "MAIN" routine where the program starts. This routine calls an initialisation routine (see below) and coordinates the animation.

(c) An "Initialisation" routine which sets up the state in its initial condition.

(d) A subroutine for each operation which performs the operation's defined function. A subroutine has:-

- (i) a unique name;
- (ii) a list of input (GIVEN) and output (YIELDING) variables;
- (iii) a list of declarations of the input and output variables and any temporary variables needed;
- (iv) a body which implements the operation's defined function.

7.3.2 Operation translation.

The most intricate part of the translation is the production of SIMSCRIPT code to implement the operation's defined function. Each operation is translated as a separate SIMSCRIPT operation. The

main parts of this process are:-

* Forming a SIMSCRIPT operation header to mimic the operation signature.

* Writing code to initialise temporary variables to hold the initial values of external "wr" values.

* Translating the pre-condition expression into an IF ... THEN ... ELSE statement.

* Translating the post-condition expression into a structured program block which yields values for free variables. These values are such that the post-condition expression evaluates to true.

* Translating the decomposition of an operation into the appropriate control structures and operation calls.

Many of the other processes are essentially an enhanced transliteration process whereby VDM symbols are replaced directly with SIMSCRIPT equivalents. The most difficult part of this process is the post-condition translation.

7.3.3 Post-condition translation.

The post-condition translation involves the manipulation of the parse tree. The aim is to sort variables into bound and free types. Bound variables are those whose value is fixed throughout the operation, i.e. input variables, read only external variables and initial values of read/write external variables. Free variables are all other variables, i.e. output variables and final values of read/write external variables. The post-condition expression is further simplified and sorted according to the type of variables contained in each sub-expression. Sub-expressions are classified into five types. These are:-

* Simple bound expressions. These are equivalences between bound variables and other bound variables or bound variables and values, e.g. in an operation containing the following declaration:-

    wr InputValve : Valve

    the initial value of the variable InputValve (InputValve') is a bound variable and hence the following part of the post-condition is a Simple bound expression:-

    post InputValve' = OPEN ...

* Complex bound expressions. These are combinations of simple bound expression using the logical "and" and logical "or" operators, e.g.

```
wr InputValve : Valve
post    (       InputValve' = OPEN
        or      InputValve' = CLOSED ) ...
```

* Simple free expressions. These are equivalences between free and bound variables or free variables and values, e.g.

```
wr InputValve : Valve
post InputValve = CLOSED ...
```

* Complex free expressions. These are combinations of simple bound expressions using the logical "and" and logical "or" operators, e.g.

```
wr InputValve : Valve
post    (       InputValve = OPEN
        or      InputValve = CLOSED ) ...
```

* Mixed expressions. These are combinations of bound expressions with free expressions using the logical "and" and logical "or" operators, e.g.

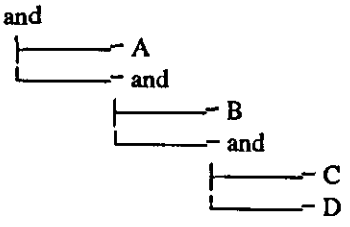
```
wr InputValve : Valve
post            (       InputValve' = OPEN
                and     InputValve = CLOSED )
        or      (       InputValve' = CLOSED
                and     InputValve  = OPEN )
```

Most post-condition expressions are mixed expressions.

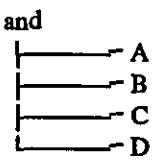
After the initial parsing of the specifications, post-condition expressions are stored as binary tree. For example the expression:

A and B and C and D

is stored as:-



In order to make translation possible it is necessary to sort the nodes according to their type. This is done by converting the binary tree into a multi-branch or n-ary tree. The above expression after conversion would be stored as a single node with four branches, i.e.



Each of the five node type is given a priority:-

5 Simple bound
4 Complex bound
3 Mixed
2 Complex bound
1 Simple bound

The branches at a single node are sorted according to their priority; 5 is highest and 1 lowest.

After this manipulation has occurred the modified post-condition parse tree is translated into the animation SIMSCRIPT code. The ultimate aim is to produce an SIMSCRIPT code to find a final state which satisfies the post-condition. Formally the final state must satisfy the following conditions:-

for an initial state : $\sigma_i \in \Sigma$

the final state : $\sigma_f \in \Sigma \cdot$ post-Op(σ_i,σ_f)

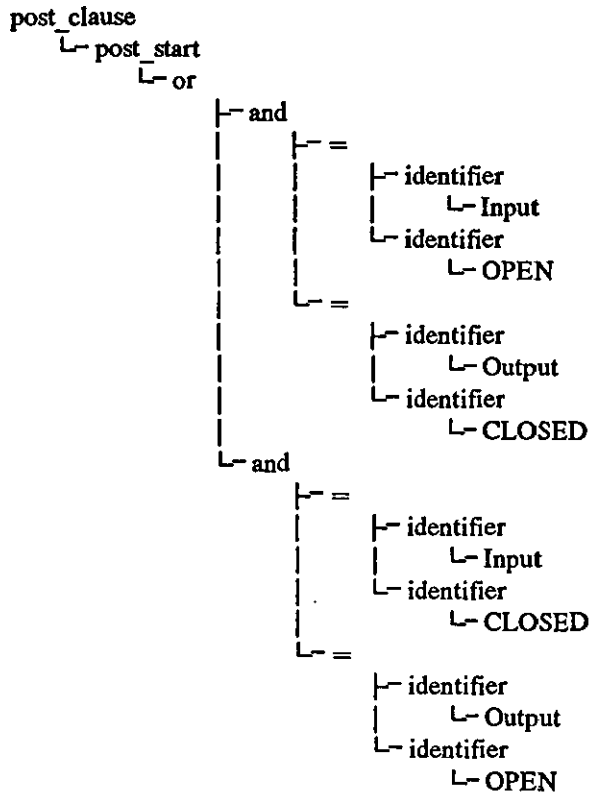
The tactics employed in the translation can be seen in the following example post-condition translation.

The operation is:-

```
Valve = { OPEN , CLOSED }
Valve_Test ( Input : Valve ) Output : Valve
ext
pre true
post    (       Input = OPEN
                and     Output = CLOSED )
        or      (       Input = CLOSED
```

```
                    and     Output = OPEN )
    endpost
    endop
```

The appropriate section of the parse tree dealing with the post-condition is:-



The post condition is translated into SIMSCRIPT code as follows. The post-condition consists of a disjunct expression. Because of the mutual exclusion requirement either one or the other, but not both, must be true. So each is tested separately. The first sub-expression is a conjunct expression. It has two parts a bound and a free part. The truth value of the bound part is defined by the initial conditions. It is therefore used as a test to see if this particular sub-expression contains the solution to the post-condition. If it is true then the final value required for the free part to be true is known and this value is set. The same process is followed for the second disjunct sub-expression. The resultant SIMSCRIPT code is:-

```
IF ( Input = ..OPEN )
        LET Output = ..CLOSED
ENDIF

IF ( Input = ..CLOSED )
```

```
        LET Output = ..OPEN
    ENDIF
```

In more complex expressions this process of using bound expressions to find values for free expressions is used repeatedly and recursively. This process is sufficient to find solutions to deterministic post-conditions. If there are non-deterministic sub- expressions in post-condition then the translator provides skeleton code for a choice between the possible alternatives. For example, in the operation:

```
ValveTest2 ( Input : Valve ) Output : Valve
ext
pre true
post    (       Input = OPEN
                and     Output = CLOSED )
        or      (       Input = CLOSED
                and (   Output = OPEN
                        or      Output = CLOSED ) )
endpost
endop
```

The post-condition is translated as:-

```
IF ( Input = ..OPEN )
        LET Output = ..CLOSED
ENDIF

IF (    Input = ..CLOSED )
        SELECT CASE .....
                CASE ...
                        LET Output = ..OPEN
                CASE ...
                        LET Output = ..CLOSED
        ENDSELECT
ENDIF
```

7.4 Identification and Implementation of SIMSCRIPT Routines to Support Graphics Output.

7.4.1 Use of SIMSCRIPT graphical displays.

There are two main displays used for the animation. There are:-

* the main display. This shows all the information about the current value of the state variables.
* the local display. Operations with input and output parameters need an additional display which shows the value of those variables.

The automatic code generator produces outline routines for updating both types of display.

7.4.2 Updating the main display.

A subroutine is generated which contains the outline SIMSCRIPT code to update the main display. This routine is based on an automatic analysis of the fields which compose the state and the possible values of these fields.

7.4.3 Updating local displays.

A subroutine is generated for each operation with input and output arguments. This routine contains the outline SIMSCRIPT code to update the display of these variables. It is again based on an automatic analysis of the possible values of these variables.

8 DEMONSTRATION ANIMATIONS.

8.1 Specification and Animation of a Simple Logic Gate.

8.1.1 The purpose of the specification.

The purpose of this example is to show clearly the stages involved in building and animating a formal specification. The systems to be specified is a simple logic gate. The function of the gate is to produce an output which is the logical "AND" of its two inputs. The logic gate is informally specified as follows:-

* The logic gate has two input lines and one output line.
* These lines may be either HIGH or LOW.
* If both input lines are HIGH then the output shall be HIGH.
* In other cases the output line should be LOW.

8.1.2 Building the specification.

An analysis of the problem leads to the following decisions about the form of the specification:

(a) There is no need for a state variable.

(b) One data type drawn from a set, called Logic, containing the elements High and Low, is needed. This is written as:-

Logic = { High , Low }

(c) The operation, called And_Gate, has two input arguments "Input1" and "Input2" both of type "Logic". There is one output argument "Output" also of type "Logic". The operation' signature is thus written as:-

And_Gate (Input1 : Logic ; Input2 : Logic) Output : Logic

(d) There are no effects on external variables. This is written as:-

ext

(e) The function is to be defined over the complete input range. This is written as:-

pre true

(f) The output is related to the inputs such that if both inputs are High the output is High, if either input is Low the output is Low. In order to specify these two separate cases the post condition is written as a disjunct between two mutually exclusive sub-expressions. The first sub-expression covers the case where the output is "High". The second covers the case where the output is "Low". This is written as:-

```
post        (     Input1 = High
            and   Input2 = High
            and   Output = High )
      or    (     (     Input1 = Low
                  or    Input2 = Low )
            and   Output = Low )
```

The above statements form the specification of the operation. The complete VDM specification is also shown in Appendix G. As there is only a single operation the only proof necessary in this case is the implementability proof. That proof for this operation is shown in Appendix D.

8.1.3 The stages of the animation prototyping process.

After prepartation, the specification is entered as a JSP-TOOL diagram. It is then processed via JT-DOC and TP2 to produce the basic specification text. This text is then passed to the animator which produces the basic animation code. The complete animation code produced by the automatic code generator is shown in Appendix G. The main routines produced by the code generator are:-

-Routine-	-Purpose-
PREAMBLE	SIMSCRIPT Data declarations.
MAIN	Where all SIMSCRIPT programs begin.
And_Gate	Implementation of the AND gate's specified function.

Tim.Local.Update.And_Gate	Updates the display of And_Gate variables
Tim.Update.Display	Updates the main display of operation name and condition type.

Finally the graphics and user interaction routines are added. In order to add graphics and user interaction a number of routines must be added. These are:-

-Routine-	-Purpose-
TIM.AND.INPUT	Get input values from user
AND.INPUT.CONTROL	Control form used for input
TIM.CHOOSE.REPEAT	Allows user to choose a further demo

The following three figures (Figure 20, Figure 21, Figure 22) show the displays used in the animated prototype of this specification.

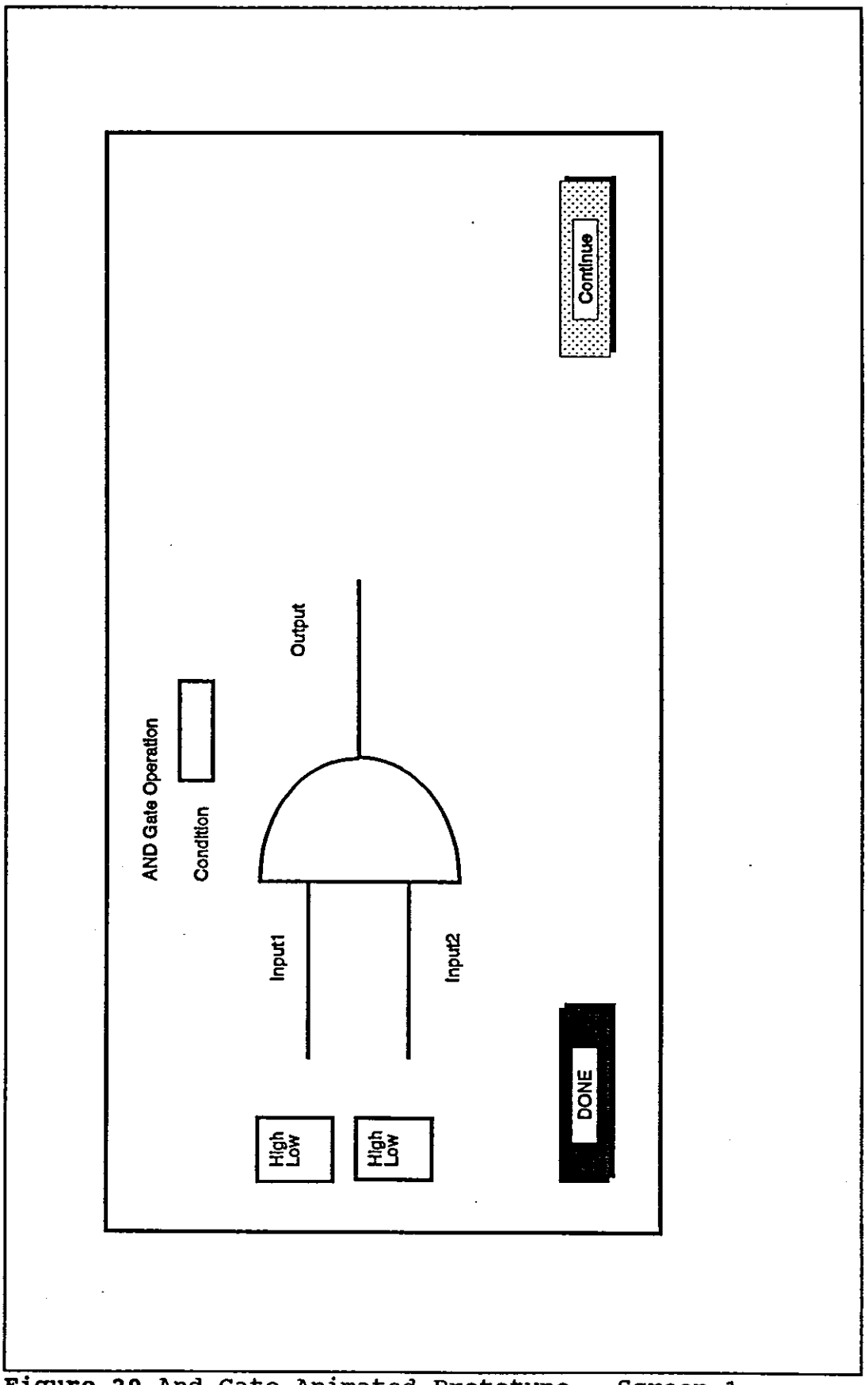


Figure 20 And_Gate Animated Prototype - Screen 1.

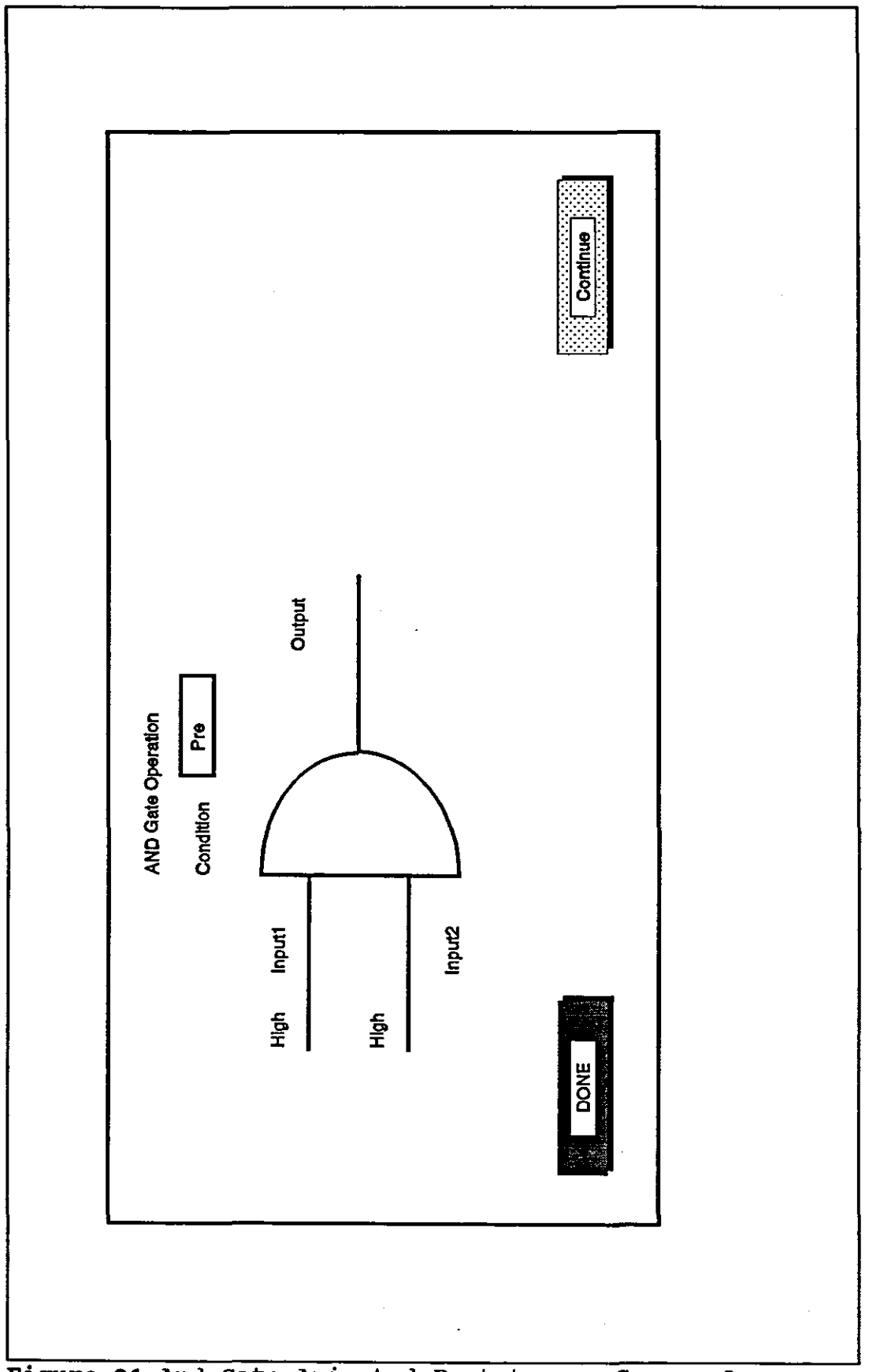


Figure 21 And_Gate Animated Prototype - Screen 2.

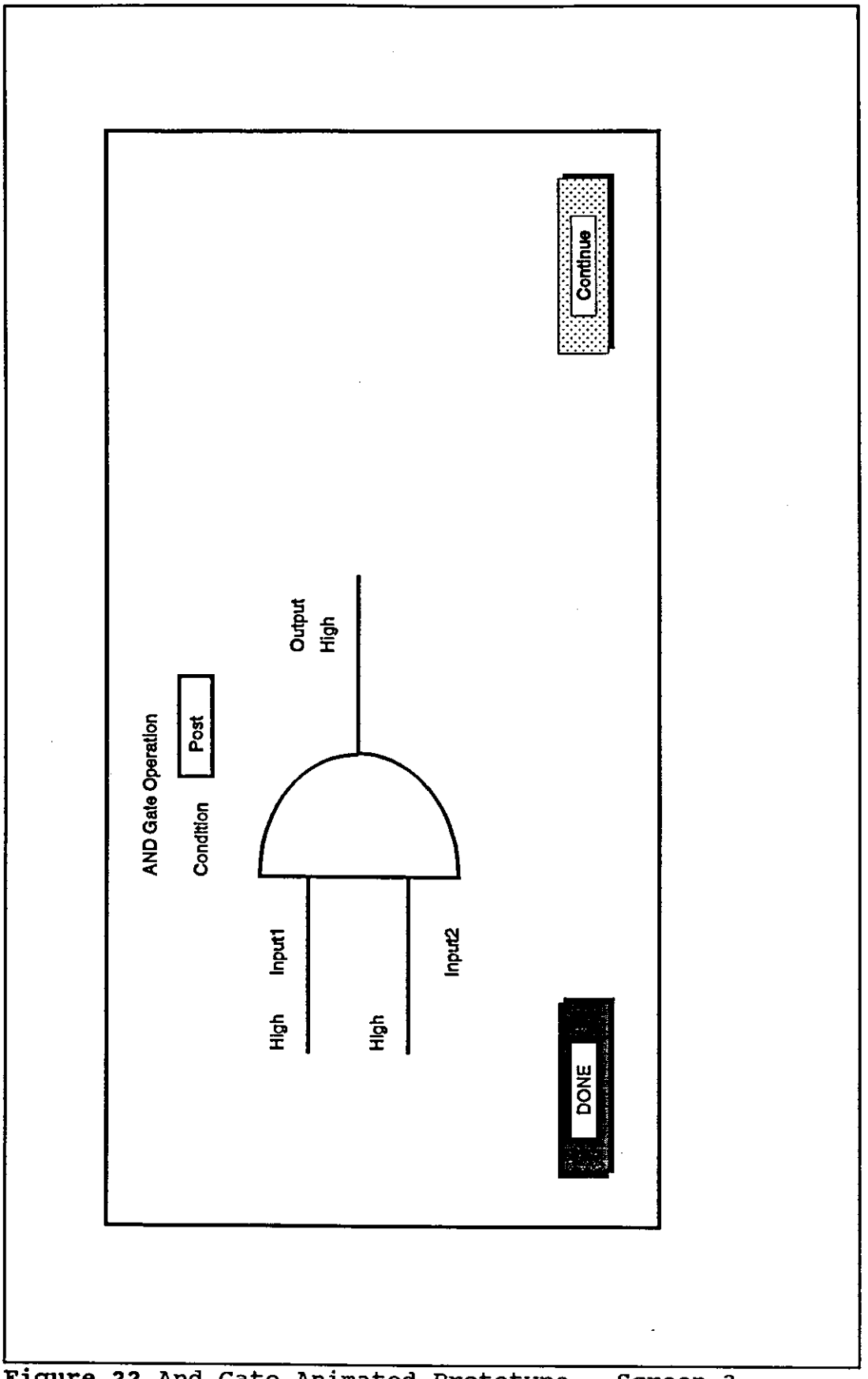


Figure 22 And_Gate Animated Prototype - Screen 3.

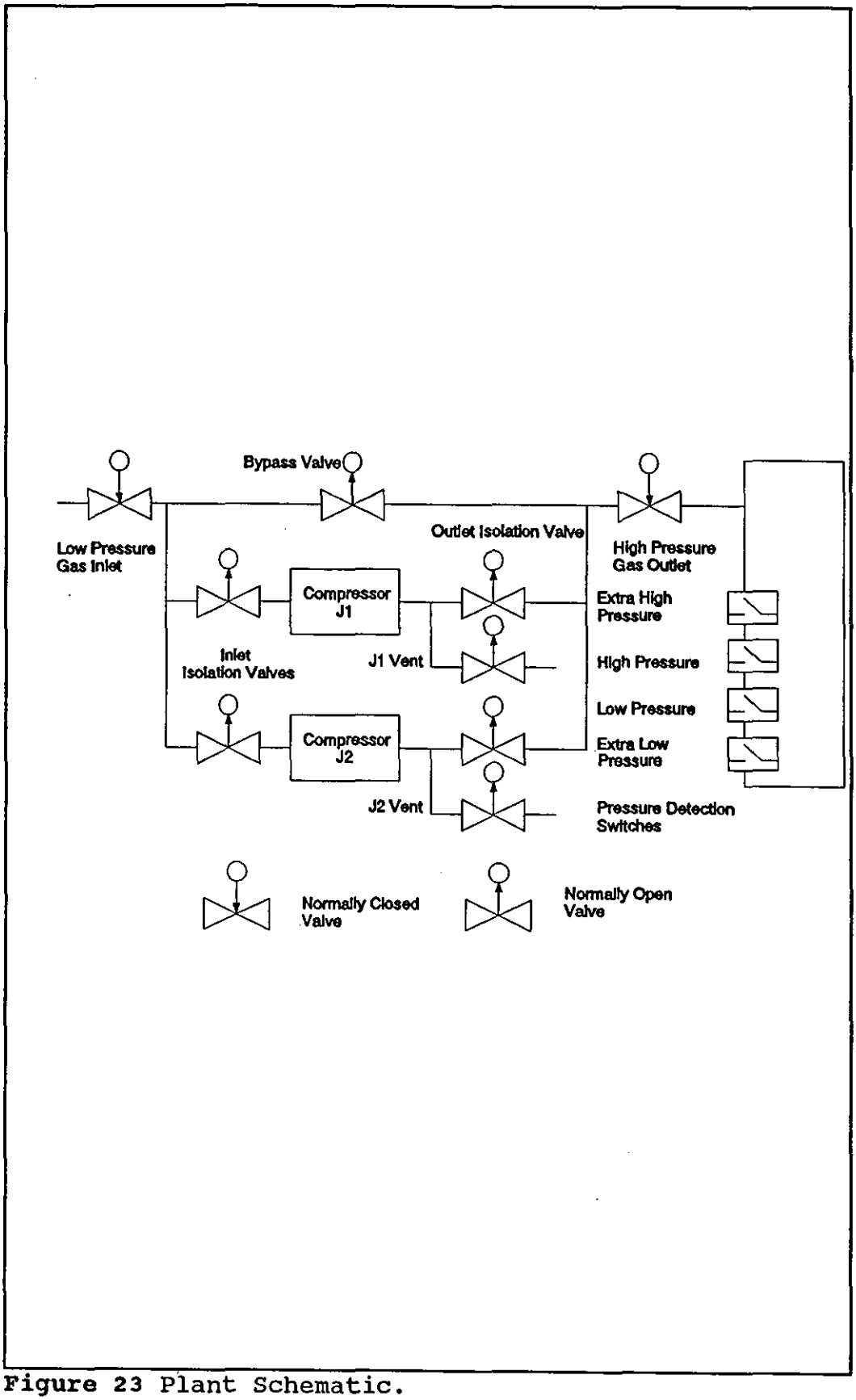


Figure 23 Plant Schematic.

8.2 Specification and Animation of a Plant Controller.

8.2.1 The purpose of the specification.

The purpose of this section is to shown the application of the animation prototyping process to a more complex, practical system. The specification of this system shows the use the formal notation defined earlier to describe more complex pre and post-conditions. Also, the decomposition of operations into sequences, selections and iterations is shown. Finally the automated production of animation code is shown.

8.2.2 The formal specification of the system.

The system described here is a large chemical plant (shown in Figure 23). Its purpose is to compress nitrogen. The compressed nitrogen is then stored in liquid form in a storage vessel. Nitrogen is drawn from the storage vessel for use in other processes. The system consists of two compressors, a storage vessel, a number of valves and a number of alarms. The compression of gas is done cyclically using first one compressor and then the other. The statement of requirements are briefly outlined in Appendix H.

These requirements were examined and the variables to describe the state of the plant were formed. Some simplification of the state variables were made. Valves and alarms with overrides are represented as a single variable. Also compressors and valves are represented as ON or OFF and OPEN or CLOSED respectively. The final model arrived at is given in appendix H.

The fail safe condition of the plant are an important requirement, therefore the plant specification begins with the operation PlantSequencer. This operation states that the plant must begin and end in the fail-safe condition.

This operation was then decomposed into three sub-operations and one of those sub-operations was further decomposed. The structure of the specification is shown in Figure 24. The full listings of the VDM text used to describe the PlantSequencer operation and its sub-operations are again given in Appendix H.

The operation StartUpPlant describes the start up of the plant when there is no high pressure gas in the system. There are three possible outcomes of this operation. These are represented by the three disjuncts in the post condition. They three cases are:-

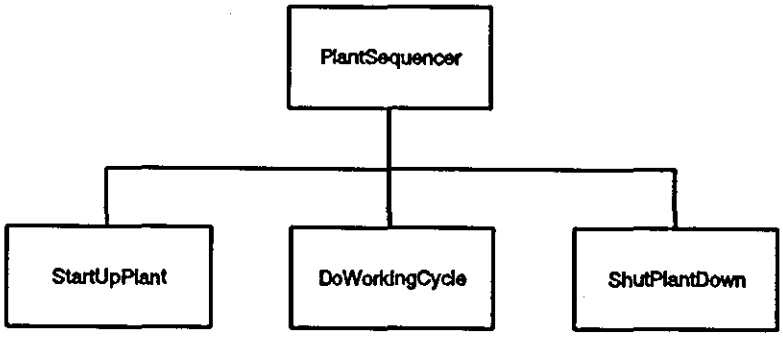


Figure 24 Structure Diagram of the Plant Specification.

(a) The first compressor (Compressor1) starts successfully and begins to compress gas correctly. The other compressor (Compressor2) remains switched off and isolated from the high pressure gas.

(b) The first compressor fails to start or compress gas correctly and is switched off, isolated and has its Lock flag set. The second compressor then starts successfully and begins to compress gas correctly.

(c) Both compressors fail in some way and are switched off, isolated and have their Lock flags set.

The operation DoWorkingCycle describes the normal working cycle of the plant during which gas is compressed using alternate compressors. There are four possible outcomes of this operation:-

(a) At some time during the cycle both compressors fail. They are switched off and isolated.

(b) The operator switch is moved to the OFF position. Both compressors must be switched off and isolated.

(c) The extra high pressure alarm begins to ring. This is an exceedingly dangerous condition. Both compressors must be switched off and isolated.

(d) Both compressors failed during StartUp. The working cycle therefore does nothing.

The operation ShutPlantDown ensures that at the end of the plant operation everything is returned to its fail safe position.

8.2.3 Animation prototyping the specification.

The specification structure diagram and the associated VDM text is processed by JT-DOC and TP2 to produce the basic animation code. This code is shown in Appendix H. A number of routines to control graphical output and the user interface are added to this code to produce the final animated prototype. These routines are:-

-Routine-	-Purpose-
SET.FORMS	This routine sets up text to offer a choice between post-condition options and decomposition level. It also sets up the main display forms.
CHOOSE.LEVEL	This routine allows the user to choose the level of the decomposition of operations shown.
LEVEL1	These three routines control the order
LEVEL2	of the operations for each level of
LEVEL3	decomposition.

The outline routine Tim.Update.Display is also completed with the relevant details about the display. These details include:-

* the name of the text box for displaying the operation name.
* the name of the text box for displaying the condition type.
* the details of icons for each state variable and how to update those icons for each value of the variable.

With these routines added the animation code is compiled to form an executable SIMSCRIPT program. The graphical displays are also constructed using the SIMSCRIPT forms editor. Finally the animation prototype is ready for demonstration.

8.3 Rigorous Proofs of Specification Consistency.

8.3.1 The purpose of proofs.

When formal notations are used to specify systems it is possible to formally prove certain things about the resulting specification. There are many questions which can be posed about the mathematical meaning and correctness of a formal specification, e.g.

* Is the specification syntax correct?
* Are the defined operations implementable?
* Is the specification internally consistent?
* Is the specification a correct refinement of a higher level specification?

* What properties does a particular data type have?

The main items of interest in this work are:-

(a) Is an operation implementable? That is, for each member of the set of states which satisfies the pre-condition, is there a corresponding member of the set of states which satisfies the post-condition?

(b) Is the decomposition of an operation into sub-operations correct? This problem was outlined earlier.

This approach is different from much previous work. In the systems considered here the emphasis is on the ordering of operations as opposed to the data structure and algorithms manipulating those structures e.g. a formal definition of a stack.

As shown in Appendix D, formal proofs for very simple specifications are extremely tedious. Therefore only outlines of the main lines of reasoning of proofs are shown here. In fact this probably gives a clearer picture of what is being proven than pages of mathematical symbols. It is important to understand, however, that the soundness of such proofs can be demonstrated mathematically if doubts arise.

8.3.2 Implementability of PlantSequencer.

The implementability of PlantSequencer is readily discernible by inspection. The main lines of reasoning for this are:

In the pre-condition:

(a) The pre-condition expression is a simple conjunction between equivalence expressions.
(b) No variable is referred to repeatedly in the expression.

Therefore:

(a) There are no contradictions in the pre-condition expression.
(b) There are some members of the set of states which make the pre-condition expression true.

In the post-condition:

(a) There are no bound variables in the post-condition expression.

(b) No variable is referred to repeatedly.

(c) Equivalence expressions include only variables and members of their defined type sets.

Therefore:

(a) There are no contradictions.

(b) For all members of the set of states which satisfy the pre-condition there are some corresponding members of the set of states which satisfy the post-condition.

This is fairly straightforward as the expressions involved are quite simple. For a specification involving more complex logical expressions it becomes necessary to use mathematical manipulation of the expressions. This arises because, as variables are referred to repeatedly, the occurrence of a contradiction is much more difficult to see by inspection. The following section looks at some more complex parts of the example specification.

8.3.3 Decomposition of PlantSequencer.

As shown in Figure 24, the PlantSequencer operation is decomposed into a sequence of three sub-operations; StartUpPlant, DoWorkingCycle and ShutPlantDown.

The first question is "do all states which satisfy pre-PlantSequencer also satisfy pre-StartUpPlant?". The answer to this is yes: the two expressions are identical. So by the same reasoning as above there are some members of the set of states which satisfy pre-StartUpPlant.

In post-StartUpPlant:

(a) There are no bound variables.

(b) The expression is a disjunct of three sub-expressions.

Now consider each disjunct separately:-

(c) Each disjunct is a conjunct of simple equivalence expressions.

(d) No variable is referred to repeatedly.

(e) Equivalence expressions include only variables and members of their defined type sets.

Therefore:

(a) There are no contradictions or fallacies.
(b) There are members of the set of states for which each disjunct is true.
(c) For all members of the set of states which satisfy the pre-condition there are some corresponding members of the set of states which satisfy the post-condition.

The next question is for these members of the set of states for which pre-StartUpPlant and post-StartUpPlant are true, is pre-DoWorkingCycle also true. The answer is again yes, as pre-DoWorkingCycle is identical to post-StartUpPlant. So by the same reasoning as above there are some members of the set of states which satisfy pre-DoWorkingCycle. For these values consider the truth value of post-DoWorkingCycle:

(a) Post-DoWorkingCycle is a disjunct of two sub-expressions.
(b) The first disjunct can only be true if the initial value of Compressor(PState) is NONE.
(c) The second disjunct can only be true if the initial value of Compressor(PState) is COMP1 or COMP2.

Therefore the two disjuncts are mutually exclusive.

Consider the first disjunct:

(d) The first disjunct contains no contradictions or fallacies. (It leaves the PState unchanged.)

Consider the second disjunct:

(e) The second disjunct contains two further disjuncts.
(f) The first sub-disjunct is true if the selected compressor remains the same. In this case the compressor locks are indeterminate. The second sub-disjunct is true if finally there is no selected compressor and both compressor locks are set.
(g) The remainder of the second disjunct contains no repeated references or bound variables.

Therefore the second disjunct contains no contradictions or fallacies.

Therefore for all members of the set of states which satisfy the pre-condition there are some corresponding members of the set of states which satisfy the post-condition.

Now consider the operation ShutPlantDown. Pre-ShutPlantDown consists of two disjuncts. They are a simplified version of Post-DoWorkingCycle. Do all members of the set of states which satisfy post-DoWorkingCycle satisfy pre-ShutPlantDown?

(a) The pre-condition consists of two disjuncts.

(b) The first disjunct is true in cases where no compressor is selected. These arise from cases where both compressors fail during StartUpPlant or both compressors fail during DoWorkingCycle.

(c) The second disjunct is true when DoWorkingCycle ended with the EHP alarm on or the operator switch off.

(d) There are no other possible outcomes of the sequence StartUpPlant followed by DoWorkingCycle.

For these values consider the expression post-ShutPlantDown. This expression is identical to post-PlantSequencer and using the reasoning above: for all members of the set of states which satisfy the pre-condition there are some corresponding members of the set of states which satisfy the post-condition.

This sequence of reasoning also leads to the conclusion that the sequence:

StartUpPlant followed by
DoWorkingCycle followed by
ShutPlantDown

Is exactly equivalent to the single operation PlantSequencer.

There are many other checks for consistency which may need to be performed on a decomposition. Many of these are simple syntactic checks; others pose questions such as "do sub-operations have side-effects (as specified in their external clauses) which are consistent with those of their parent operation?". The complexity of such proofs increases with the complexity of the formal notations used. The notation used here attempts to keep things simple by avoiding the problems involved with such things as data reification and the temporal or modal logics.

9 COMMENTS AND CONCLUSIONS.

This thesis has outlined a potential weakness in the use of formal specifications for industrial projects: ensuring that such specifications accurately define the client's requirements. The ability of clients to interpret formal specifications is seen as crucially important in the development of correct software. Animation prototypes is a means of aiding this interpretation. By eliminating the need to understand the details of formal notations and their mathematical foundations, it makes formal specifications accessible to clients. This research has demonstrated the feasibility and suitability of animation prototyping from formal specification of real-time systems.

The work here has shown that a restriced set of VDM notations is sufficient to specify real-time systems whilst retaining the deductive apparatus of the formal system. This facilitates rigorous checking of specification development. The use of a rigorously provable method of design by decomposition has been shown. The subset of the VDM notation (here defined as S-VDM) includes the constructs necessary to support this approach. Structured diagramming techniques which support this decomposition method have also been defined.

Animation prototypes - that is, the use computer animated pictures - has been shown to be a powerful technique for the development of system specifications. Further, it is seen as especially advantageous when used to animate formal specifications. The building and use of custom made animation prototypes has been shown. A more cost-effective approach to prototyping is to derive prototypes directly from specifications. Therefore, a theoretical framework for deriving animated prototypes automatically from specification written in S-VDM has been developed. To demonstrate the practicality of this, software has been designed which automates the animation prototyping process.

The usefulness of S-VDM for specifying real-time systems has been demonstrated by specifying a chemical plant controller. The power of the technique of animation prototyping has been shown by using this specification as the basis for animated demonstrations. These demonstrations use code which has been automatically derived from the formal specifications using the tools and techniques described in this thesis.

10 Recommendations for Future Work.

The concept and application of animation prototyping has been clearly established in this thesis. However, a number of key areas of the process could be improved. These compromise:-

* the tool and its use.
* the formal language.
* reaoning about timing in specifications.

10.1 Improvements to tools and their use.

Improvements to the tools used in the animation process would bring a number of benefits. Areas of particular interest are:-

(a) The placement of VDM text on structure diagrams. Currently the diagramming style used is restricted by the use of the JSP-TOOL program. The building of specifications would be greatly enhanced if the VDM text associated with an operation could be stored hypertext style as described in chapter 7. Work to identify and evaluate different tools which allow greater flexibility in the style of diagram structuring should be undertaken. Further work on the animation tool to cope with this new style is seen as very useful.

(b) The linkage between the displays and the automatically generated code. This is a very important part of the animation process. There is scope for more code generation to be done automatically. A useful piece of work would be to develop the syntax of a notation to give more information to the animation tool. The notation would need to describe such things as:

* the file names of SIMSCRIPT "forms" used;
* the displays used to represent the values of fields in the state;
* the text to be displayed in places where the user is asked to choose between alternatives.

If this infromation were incorporated into the specification text file then a great deal of the outline SIMSCRIPT code which is currently generated and then completed manually could be generated entirely automatically.

(c) The declaration of local variables. The decomposition structure does not allow this. There is a need to extend the scope of the specification notation to include such declarations. Also needed are operations to manipulate the values of such variables. This will lead to more flexiblibilty in the structured ordering of sub-operations in a decomposition.

(d) The parser error reporting. In the automatic code generator, the parser's current error reporting is very basic. The process of developing syntactically correct specifications would benefit greatly from more informative error messages.

(e) Non-determinism in specifications. This must be carefully handled by the specifier. It is his responsibility to ensure that there is no implicit non-determinism. If the expression translation mechanism were more sophisticated, this property of expressions could be highlighted by the tool. This would be a very useful aid to generating better quality specifications.

10.2 Improvements to the formal language.

Useful improvements to the current specification notation would include:-

(a) Better structuring of composite data types. At present the data types and operations handled by the animation tool are very limited. The use of the same basic type but with more structuring of composite types would help a great deal in the construction of larger and more comprehensible specifications. This limitation arises mainly through the use of the SIMSCRIPT language which has only a very limited number of basic data types avaliable. With SIMSCRIPT, in order to incorporate this, the tool would need to generate special operations to check for equivalence between two composite variables. It would also need to produce special routines to assign the field values of one composite variable to the fields of another. MODSIM (also manufactured by CACI) is a new system which supports the same graphical features as SIMSCRIPT. However, it has a much more modern style programming language. Converting the automatic code generator to MODSIM would represent a useful next step.

(b) The use of number-based types. The current emphasis of the specification language is biased towards enumerated data types suitable for describing state transitions in systems. The translation in post-condition expressions of operations such as $>, <, <=$ and $>=$ and the representation in animations of the large number of possible solutions which arise from their

use would be a valuable piece of work.

(c) The use of negation in logical expressions. In logical expressions, negation is not yet supported. It would be relatively straightforward to introduce the operators "NOT" and "NOT.EQUAL" into expressions involving only bound quantities i.e operation pre-conditions, selection and iteration conditions. However, more work needs to be done on their use in those expressions which involve free variables i.e post-conditions and initial states. Again translation and the presentation of such information in an animation would be central to any solution of this problem.

10.3 More reasoning about time.

The incorporation of descriptions of time and event dependencies in specifications is very desirable. A number of notations and theories for reasoning about temporal aspects of specifications have emerged. The incorporation of temporal notations into animatable specifications may now be possible. Defining a new notation to incorporate temporal reasoning or using an existing notation, such as MAL, would provide a very challenging piece of work for a future research.

REFERENCES.

Adhami88 — Adhami,E., Shand,J. and McNeile,A., "An Environment for the Execution and Graphical Animation of a JSD Specification," Second IEE/BCS Conference: Software Engineering 88 (Liverpool, UK, 11-15th July 1988), IEE, London, UK, 1988, pp.138-142.

Alavi84 — Alavi,M., "An Assessment of the Prototyping Approach to Information Systems Development," Communications of the ACM, Vol.27, No.6, June 1984, pp.556-563.

ALVEY84 — Alvey programme - Software Engineering Document - Programme for formal methods in system development, Alvey Directorate, London, April 1984.

Baldassari88 — Baldassari,M., Barti,V., Bruno,G., "Object-oriented conceptual programming based on PROT nets", Proceedings of the 1988 International Conference on Computer Languages (Miami Beach, FL, USA, 9-13 Oct 1988), IEEE Computer Society Press, Washington, DC, USA, 1988, pp.226-233.

Balzer83 — Balzer,R., Cheatham,T., Green,C., "Software Technologies in the 1990's: Using a New Paradigm," IEEE Computer, Vol.16, No.11, Nov 1983, pp.39-45.

Berzins88 — Berzins,V., "Object-Oriented Techniques Based on Specification," Proceedings COMPSAC 88 : The 12th International Computer Software and Applications Conference (Chicago, IL, USA, 5-7 Oct 1988), IEEE Computer Society Press, Washington, DC, USA, 1988, pp.437-438.

Bjorner82 — Bjorner,D. and Jones,C.B., "Formal Specification and Software Development", Prentice Hall International, Inc., 1982.

Bjorner87 — Bjorner,D., Jones, C.B., Mac an Airchinnigh,M. and Neuhold,E., eds, "VDM - A Formal Method at Work", Proceedings VDM-Europe

Symposium, Lecture Notes in Computer Science, vol.252, 1987.

Bloomfield88 Bloomfield,R., Marshall,L. and Jones,R., eds, "VDM - The Way Ahead", Proceeding 2nd VDM-Europe Symposium, Lecture Notes in Computer Science, vol.328, Springer Verlag, 1988.

Blumofe88 Blumofe,R. and Hecht,A., "Executing Real-time Structured Analysis Specifications." ACM SIGSOFT: Software Engineering Notes (USA), Vol.13, No.3, July 1988, pp.32-40.

Boehm76 Boehm,B.W., "Software Engineering.", IEEE Transactions on Computers, vol 25, no.12, Dec 1976, pp.1226-1241.

Boehm88 Boehm,B.W., "A spiral model of software developement and enhancement", Computer, vol.21, no.5, May 1988, pp.61-72.

Booch86 Booch,G., "Object Oriented Development," IEEE Transactions on Software Engineering, Vol.12, No.2, Feb 1986, pp.211-221.

Bornat79 Bornat,R., "Understanding and Writing Compilers", MacMillan Publishers Ltd, Houndsmill, Basingstoke, Hampshire, 1979.

Brookes87 Brookes,F.P. Jr., "No Silver Bullet: Essence and Accident of Software Engineering," IEEE Computer, Vol.20, No.4, April 1987, pp.10-19.

Brinksma88 Brinksma,E., ed. "Information Processing Systems - OSI - LOTOS - A Formal Technique Based on Temporal Ordering of Observational Behaviour", ISO IS 8807/1988.

Bruno86 Bruno,G. and Balsamo,A., "Petri net-based object-oriented modelling of distributed systems", Proceedings of the ACM Conference on Object-oriented Programming (Portland, Oregon, USA October 1986), 1986, pp.184-293.

BSI89 BSI, "VDM Specification Language: Proto-Standard", IST/5/50, 1989.

Budde84 Budde,R., Kuhlenkamp,K., Mathiassen,L. and Zullighoven,H. eds ,

"Approaches to Prototyping," Springer Verlag, 1984, ISBN 3-540-13490-5.

Burstall80 — Burstall,R.M. and Goguen,J.A., "The semantics of Clear, a specification language.", In Bjoner,D., ed., Absract Software Specification, Lecture Notes in Computer Science, vol.86, Springer Verlag, 1980, pp.292-332.

Cameron86 — Cameron,J.R., "An overview of JSD", IEEE Transactions on Software Engineering, Vol.12, No.2, February 1986, pp.222-240.

Carlow84 — Carlow,G.D., "Architecture of the space shuttle primary avionics software.", Commun. ACM, vol27, no.9, Sept 1984, pp.926-936.

Clarke90 — Clarke,R.G., "The design and development of embedded Ada systems", Software Engineering Journal, May 1990, pp.175-184.

Cohen82 — Cohen,B., "Justification of formal methods for system specification", Software and Microsystems, Vol.1, No.5, August 1982, pp.119-127.

Cooling89 — Cooling,J.E and Hughes,T.S., "The emergence of rapid-prototyping as a real-time software development tool", Proceedings 2nd IEE/BCS International Conference on Software Engineering, Sept.1989, pp.60-64.

Coomber90 — Coomber,C.J. and Childs,R.E., "A graphical tool for the prototyping of real-time systems", ACM SIGSOFT Software Enginnering Notes, vol.15, no.2, April 1990, pp.70-83.

CORE86 — "CORE - Controlled Requirements Expression", System Designers plc, Fleet, Hampshire, GU13 8PD, document no. 1986/0786/500/PR/0158.

Davis89 — Davis,R.E.,"Truth, Deduction, and Computation: logic for computer science", W.H.Freeman and Co., NY, USA, 1989.

DeMarco78 — DeMarco,T., "Structured Analysis and System Specification," Yourdon Press, New York, NY, USA, 1978.

Dearnley83 — Dearnley,P.A. and Mayhew,P.J., "In Favour of Systems Prototypes and Their Integration into the Systems Development Cycle," Computer Journal, Vol.26, No.1, Jan 1983, pp.36-42.

DEF STAN 00-55 — Defence Standard 00-55. UK Ministry of Defence standard for the procurement and use of software for safety critical applications, HMSO, 1989.

Denvir86 — Denvir,T., "Introduction to Discrete Mathematics for Software Engineering", MacMillan Education Ltd, London, UK, 1986.

Diaz-Gonzalez88 — Diaz-Gonzalez, J.P. and Urban, J.E., "Language aspects of ENVISAGER: an object-oriented specification environment for real-time systems", Proceedings of the 1988 International Conference on Computer Languages (Miami Beach, FL, USA, 9-13 Oct 1988), IEEE Computer Society Press, Washington, DC, USA, 1988, pp.214-225.

Diaz-Gonzalez89 — Diaz-Gonzalez, J.P. and Urban, J.E., "Prototyping conceptual models of real-time systems: a visual perspective", Proceedings of the 22nd Hawaii Conference on Systems Science (Hawaii, USA, 3-6 January 1989), Vol2, IEEE Computer Society Press, 1989, pp.358-67.

Dreyfus86 — Dreyfus.H.L. and Dreyfus,S.E., "Mind Over Machine," Blackwell, Oxford, UK, 1986, ISBN: 0-631-15126-5.

Ehrig85 — Ehrig,H. and Mahr,B. "Fundementals of Algebraic Specification 1", Springer Verlag, 1985.

Ericcson84 — Ericcson,A. and Simon,H.A., "Protocol Analysis, Verbal Reports as Data," The MIT Press, 1984.

Fathi84 — Fathi,E.T. and Fines,N.R., "Real-time data acquisition, processing and distribution for radar applications.", Proc IEEE 1984 Real-Time Systems Symp., Dec 1984, pp.95-101.

France89 — France,R.B., "Formal specification using structered systems analysis", Proceedings of ESEC '89 - 2nd European Software Engineering

Conference, Coventry, UK, 11-15 September 1989, Springer Verlag, Berlin, pp.293-310.

Galton90 — Galton,A., "Logic for Information Technology", J.Wiley and Sons, NY, USA, 1990.

Gibbons87 — Gibbons,P.F., "What are formal methods?", Information and Software Technology, Vol.30, No.3, April 1988, pp.131-137.

Goldsack88 — Goldsack,S.J., "Specification of an operating system kernel - FOREST and VDM compared", VDM 88, Proceedings 2nd VDM-Europe Symposium, Dublin, Ireland, Sept.1988, pp.88-100.

Gomma81 — Gomaa,H. and Scott,D.B.H., "Prototyping as a Tool in the Specification of User Requirements," Proceedings of the 5th International Conference on Software Engineering, IEEE Computer Society Press, New York, NY, USA, 1981, pp.333-338.

Goguen79 — Goguen,J.A. and Tardo,J. "An Introduction to OBJ: A Language for Writing and Testing Software", in Proceeding of the Conference on Specification of Reliable Systems, IEEE Computer Society Press, 1979, pp.170-189.

Goguen82 — Goguen,J.A. and Meseguer,J., "Rapid prototyping in the OBJ executable specification language", ACM SIGSOFT Software Engineering Notes, vol.7, no.5, Dec, 1982, pp.75-83.

Goguen84 — Goguen,J.A., "Parameterised programming", IEEE Transactions on Software Engineering, vol.10, no.5, 1984, pp.528-544.

Guttag85 — Guttag,J.V., Horhing,J.J. and Wing,J.M. "Larch in Five Easy Pieces", Digital Systems Research Center Report, July 1985.

Guttag86 — Guttag,J.V. and Horning J.J. "Report on the Larch Shared Language", Science of Computer Programming, vol.6, 1986, pp.103-134.

Hall90 — Hall,A., "Seven Myths of Formal Methods", IEEE Software, Vol23,

No9, Sept 1990, pp.11-19.

Harel90 — Harel,D., Lachover,H. et al., "STATEMATE: A working environment for the development of complex reactive systems", IEEE Transactions on Software Engineering, vol.16, no.4, April 1990, pp.403-414.

Hatley88 — Hatley,D. and Pirbhai,E., "Strategies for real-time systems specification", Dorset Publishing House, 1988.

Hayes87 — Hayes,I.(ed), "Specification Case Studies", Prentice Hall, 1987.

Hekmatpour86 — Hekmatpour,S. and Ince,D.C., "Formal Specification-Based Prototyping System," in Software Engineering 86, eds D.Barnes and P.Brown, IEEE Computing Series Vol.6, ISBN: 0-86341-082-0, pp.317-335.

Hekmatpour86 — Hekmatpour,S. and Ince,D.C.,"Software Prototyping, Formal Methods and VDM", Addison-Wesley, 1988.

Henderson86 — Henderson,P., "Functional programming, formal specification and rapid prototyping", IEEE Transactions on Software Engineering, vol.12, no.2, 1986, pp.241-250.

Hoare85 — Hoare,C.A.R., "Communicating Sequential Processes", Prentice Hall International, 1985.

Hughes87 — Hughes,J.M.L. and Prescott,I.C. "Automation - What of the Future?", Proceedings 8th International Ship Control Symposium Vol.1, pp.67-80.

Johnson87 — Johnson.P.E., Zualkernan,I. and Garber,S., "The Specification of Expertise," International Journal of Man Machine Studies, 26, 1987, pp.161-181.

Jones88 — Jones,G. and Prieto-Diaz,R., "Building and Managing Software Libraries," Proceedings COMPSAC 88 : The 12th International Computer Software and Applications Conference (Chicago, IL, USA,

5-7 Oct 1988), IEEE Computer Society Press, Washington, DC, USA, 1988, pp.228-236.

Jones90 — Jones,C.B., Systematic software development using VDM, Second Edition, Prentice-Hall International, 1990.

Jordan89 — Jordan.P.W, Keller,K.S., Tucker,R.W. and Vogel,D., "Software Storming: Combining Rapid Prototyping and Knowledge Engineering," IEEE Computer, Vol.22, No.5, May 1989, pp.39-48.

Kaplan85 — Kaplan,G., "The X-29: Is it coming or going?", IEEE Spectrum, June 1985, pp.54-60.

Kelton84 — Kelton,P.W., "Distributed computing for astronomical data acquisition at McDonalds Observatory.", ibid, pp.83-88.

Kemmerer85 — Kemmerer,R.K., "Testing Formal Specifications to Detect Design Errors," IEEE transactions on Software Engineering, Vol.11, No.1, Jan 1985, pp.32-43.

Kramer88 — Kramer,J., Ng,K., Potts,C. and Whitehead,K., "Tool Support for Requirements Analysis," Software Engineering Journal, Vol.3, No.3, May 1988, pp.86-96.

Kreutzer90 — Kreutzer,W., "Tiny Tim - A Smalltalk toolbox for rapid prototyping and animation of models", Journal of Object-oriented Programming (USA), Vol.2, No.5, 1990, pp.27-36.

Loucopoulos89 — Loucopoulos,P. and Champion,R.E.M., "Knowledge-Based Support for Requirements Engineering," Inf. Software Technology (UK), Vol.31, No.3, 1989, pp.123-135.

Luqi88a — Luqi, Berzins,V. and Yeh,R.T., "A Prototyping Language for Real-Time Software," IEEE Transactions on Software Engineering, Vol.14, No.10, Oct 1988, pp.1409-1423.

Luqi88b — Luqi, Berzins,V., "Rapidly Prototyping Real-time Systems," IEEE

Software, Vol.5, No.5, Sept 1988, pp.25-36.

Luqi88c — Luqi, Berzins,V., "Execution of a High-Level Real-Time Language," Proceedings of Real-time Systems Symposium (Huntsville, AL, USA, 6-8 Dec 1988), IEEE Computer Society Press 1988, pp.67-72.

Luqi88d — Luqi, "Knowledge-Based Support for Rapid Software Prototyping," IEEE Expert, Vol.3, No.4, November 1988, pp.9-18.

Luqi90 — Luqi, "Automated prototyping and data translation", Data and Knowledge Engineering, Vol.5, No.2, North-Holland 1990, pp.167-77.

McCracken82 — McCracken,D.D. and Jackson,M.A., "Life Cycle Concept Considered Harmful," ACM SIGSOFT Software Engineering Notes, Vol.7, No.2, April 1982, pp.29-32.

Maibaum86 — Maibaum, T.S.E., Jeremaes,P. and Khosla,S., "A Modal (Action) Logic for Requirements Specification", Brown, P.J. and Barnes,D.J., eds., "Software Enginnering '86", Peter Peregrinus, 1986.

Mason83 — Mason,R.E.A. and Carey,T.T., "Prototyping Interactive Infomation Systems," Communications of the ACM, Vol.26, No.5, May 1983, pp.347-354.

Maude91 — Maude,T. and Willis,G., "Rapid Prototyping: the management of software risk", Pitman, London, 1991.

Meyer85 — Meyer,D, "On formalism in specification", IEEE Software, Vol.2, No.1, Jan 1985, pp.6-26.

Milner80 — Milner,R., "A Calculus of Communicating Systems", Lecture Notes in Computer Science, vol.92, Springer Verlag, 1980.

Musa85 — Musa,J.D., "Software Engineering: The Future of a Profession." IEEE Software Vol.2, No.1, Jan 1985, pp.55-62.

Nelson81 — Nelson,V.P. and Fellows,H.L., "A microcomputer-based controller for

an amusement park ride.", IEEE Micro, Aug 1981, pp.13-22.

ObjEx90 "ObjEx User Reference Manual", Gerrard Software, 24 Duke Street, Maccelsfield, Cheshire, U.K., 1990.

Pacini87 Pacini,G.P. and Turini,F., "Animation of Requirements Specification," in Industrial Software Technology, ed R.Mitchel, IEE Computing Series Vol.10, Peter Peregrims 1987, ISBN: 0-86241-084-7, pp.107-121.

Pomberger84 Pomberger,G., Software Engineering and Modula-2, Prentice-Hall International (UK) Ltd, UK, 1984.

Potts88 Potts,C., "The Other Interface: Specifying and Visualising Computer Systems", in Working With Computers: Theory versus Outcome, ed G.C.van der Veer, T.R.G.Green, J-M.Hoc & D.M.Murray, Academic Press, London, 1988, pp.145-175.

Prager87 Prager,D.L. and Burke,M.M., "Dependable Software", Proceedings 8th Ship Control Systems Symposium, Vol.4, 1987, pp.25-33.

Ramamoorthy84 Ramamoorthy,C.V., Prakash,A., Tsai,W-T. and Usuda,Y., "Software Engineering: Problems and Perspectives," IEEE Computer, Vol.17, No.10, Oct 1984, pp.191-209.

Ratcliff88 Ratcliff,B., "Early and Not-so-early Prototyping - Rationale and Support," Proceedings COMPSAC 88 : The 12th International Computer Software and Applications Conference (Chicago, IL, USA, 5-7 Oct 1988), IEEE Computer Society Press, Washington, DC, USA, 1988, pp.127-134.

Reeves90 Reeves,S. and Clarke,M., "Logic for Computer Science", Addison-Wesley Publishers Ltd, Wokingham, UK, 1990.

Royce70 Royce,W.W., "Managing the development of large software systems: concepts and techniques.", Proc WESCON, August 1970. Reprinted in: Proceedings of the 9th International Software Engineering Conference,

Monterey, CA, USA, IEEE Computer Society Press, 1987.

Sandberg89 — Sandberg,D.W., "Smalltalk and exploratory prototyping", ACM SIGPLAN Notes (USA), Vol.23, No.10, 1989, pp.85-92.

Schneider87 — Schneider,R.J., "Prototyping Toolsets and Methodologies: User/Developer Sociology," Proceedings of 1987 International Conference on Systems, Man and Cybernetics, IEEE, New York, NY, USA, Vol.1, 1987, pp.208-216.

Smith88 — Smith,S.L. and Gerhart,S.L., "STATEMATE and Cruise Control: A Case2 Study," Proceedings COMPSAC 88 : The 12th International Computer Software and Applications Conference (Chicago, IL, USA, 5-7 Oct 1988), IEEE Computer Society Press, Washington, DC, USA, 1988, pp.49-56.

Smullyan88 — Smullyan,R. "Forever undecided: a puzzle guide to Gödel", Oxford University Press, 1988.

Snodgrass88 — Snodgrass,J.G. and Yun,.D.Y.Y., "Software requirements specification from a cognitive psychology perspective", Proceedings 1988 International Conference on Computer Languages (Miami Beach, FL, USA, 9-13 Oct 1988), IEEE Computer Society Press, Washington, DC, USA, 1988, pp.422-430.

Spivey89 — Spivey,J.M., "The Z Notation: a reference manual", Prentice Hall, 1989.

St-Denis90 — St-Denis,R., "Specifcation by example using graphical animation and a production system", Proceedings 23rd Annual Hawaii International Conference on Systems Science (Kailu-Kona, Hawaii, USA, 2-5 Jan 1990), IEEE Computer Society Press, Los Alamitos, CA, USA, 1990, Vol.2, pp.237-246.

Swartout82 — Swartout,W. and Balzer,R., "The Inevitable Intertwining of Specification and Implementation," Communications of the ACM, Vol.25, No.7, July 1982, pp.438-440.

STARTS87 — The STARTS Guide - A guide to methods and software tools for the construction of large real-time systems, NCC publications, Manchester, 1987.

Tanik89 — Tanik, M.M. and Yeh, R.T., Rapid prototyping in software developement, IEEE Computer, Vol.22, No.5, May 1989.

Tsai88 — Tsai,J.J.P., Aoyama,M. and Chang,Y.L., "Rapid Prototyping Using FRORL Language," Proceedings COMPSAC 88 : The 12th International Computer Software and Applications Conference (Chicago, IL, USA, 5-7 Oct 1988), IEEE Computer Society Press, Washington, DC, USA, 1988, pp.410-417.

Ward85 — Ward,P.T. and Mellor,S.J., "Structured systems for real-time systems", Yourdon Press, New York, 1985.

Watkins88 — Watkins,P., "Investigation into Improvements in Software Development Centring on Rapid Prototyping," Final Year Project Report, Department of Electronic and Electrical Emngineering, Loughborough University of Technology, April 1988.

Williams84 — Williams,C.D., "The data acquisition, data reducution and control system (DARCS) for the NRCC 2x3m windtunnel.", Proc IEE 1984 Real-Time Systems Symp. Dec 1984, pp.89-94.

Willimas87 — Williams,S.J., "Nitrogen compressor plant simulator", Final Year Project Report, Dept Electronic and Electrical Engineering, Loughborough University of Technology, UK, April 1987.

Woodcock88 — Woodcock, J.C.P. and Loomes,M., "Software Engineering Mathematics", Pitman Publishing, London, UK, 1988.

Yourdon89 — Yourdon,E., "Modern structured analysis", Prentice Hall, Englewood Cliffs, New Jersey, 1989.

Zave84 — Zave,P., "The Operational Verses Conventional Approach to Software Development," Communications of the ACM, Vol.27, No.2, Feb 1984,

pp.104-118.

Zualkernan88 Zualkernan,I.A. and Tsai,W.T., "Are Knowledge Representations the Answer to Requirements Analysis?" Proceedings 1988 International Conference on Computer Languages (Miami Beach, FL, USA, 9-13 Oct 1988), IEEE Computer Society Press, Washington, DC, USA, 1988, pp.437-443

APPENDIX A.

A AN EXPERIMENT IN ANIMATION PROTOTYPING.

A.1. Introduction.

There was a need to experience some of the problems which arise during the building and use of an animation prototype. It was decided that certain issues could not be resolved without first-hand experience. The experiment was arranged with the help of a local company, Transmitton Limited.

The experiment had four phases:

(a) Establishing basic objectives.
(b) Early prototype development.
(c) Prototype refinement.
(d) Final development.

These phases and the lessons learned are set out below.

A.2. Basic objectives.

Engineers from Transmitton acted as customers wishing to use a token bus based local area network. An animation prototype was built and used to explore problems arising during building and demonstrations. This prototype formed the basis of discussions and was used to demonstrate important properties of the system.

The demonstration system was based on the IEEE 802.4 standard. This is a draft standard for access control in local area networks. This is a bus based system. The system consists of a number a stations. These stations are connected to a common communications medium. The right to transmit over the bus is controlled by the passing of a token from station to station. The order of passing forms a "logical ring". Each station has a unique logical address. The logical ring is formed such that each station has a successor whose address is lower than its own address. The station present with the lowest address is unique in that has a successor whose address is higher than its own. A typical logical

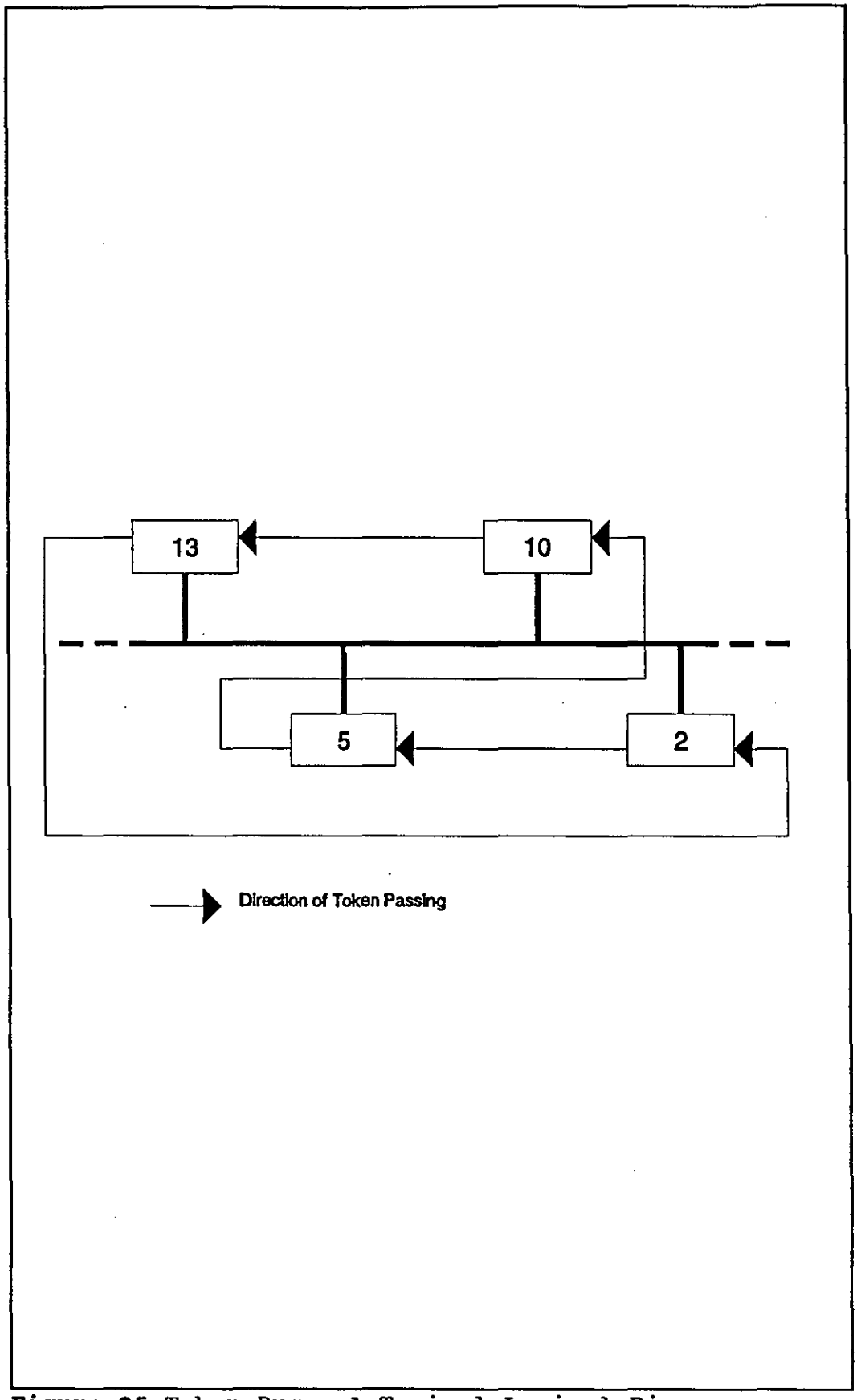


Figure 25 Token Bus - A Typical Logical Ring.

ring is shown in Figure 25. The token is thus passed from the station with the highest logical address via other stations in descending order until it reaches the station with the lowest logical address when it is passed back to the station with the highest address. The process then repeats.

The logical ring is dynamic in that stations may join and leave the ring whenever they wish. Protocols governing the addition and deletion of stations from the ring have been defined.

A.3. Early prototype development.

The first model built showed only the basic token passing. It was carefully structured to allow for more complex modelling. Objects were defined which closely reflected the real world structure of the system. These object were stations, message frames and channel sections. These objects have the following properties:

(a) Stations. These represent the stations connected to the network. They are responsible for creating message frames at the appropriate times. They also receive and respond to frames from other stations. The model of the Access Control Mechanism is contained entirely within the operation of a single station.

(b) Channel sections. These represent the physical medium connecting adjacent stations. The whole channel is made up of a number of channel sections connected together. They are responsible for simulating the propagation of message frames between stations.

(c) Message frames. These represent the packets of data which pass from station to station during the operation of the network. They are the only mechanism by which stations can detect each others presence.

The underlying model of the system and interaction in the first model were very basic and consequently the following important comments were made after the first demonstrations:

(a) Things happen too fast. There is a need to allow single stepping through the model.

(b) Concern was expressed about the ability of the token bus system to deal with the customer proposed system.

At this point it was concluded that a more detailed model of the system would be necessary in order to dispel these concerns. The following areas were targeted for further development:

(a) Start up procedure. The model should be able to demonstrate how the network behaves as stations are switched on.

(b) More details such as logical address should be user definable.

(c) Making the model more interactive. Allowing for stations to be switched on or off at any time during the simulation.

With these objectives established the model was further refined.

A.4. Prototype refinement.

The next model had a number of improvements designed to answer the questions posed above.

(a) Station addresses were made user-definable.
(b) A menu bar was added to allow for interaction with the model.
(c) The access control mechanism was partially modelled. At this stage it was unable to deal fully with the start up procedure.

After demonstration of this new model to the customers the only real comment made was about the start up mechanism. More detailed modelling of the access control mechanism was needed to resolve properly customer queries. This objective now became a top priority. A detailed modelling of the access control mechanism of the IEEE 802.4 standard was thus undertaken.

A.5. Final developement.

The final demonstration to the customers took place with the detailed model completed. This model was able to demonstrate start up, normal operation and ring maintenance fully. The comments about the model thus shifted to issues of presentation. It was generally felt that there was not enough information displayed on the screen. In particular:

(a) Both the stations logical address and its user defined name should be displayed.

(b) A key was needed on screen giving the colour and meaning of the various different protocol frames which the system uses.

(c) Frames should have directional arrows showing their direction of propagation along the bus.

(d) The access control mechanism is implemented as a state machine. Displaying the current state of a station's access control machine would be very useful.

A number of other points about the nature of the menu-based interaction were made:

(a) In the program set up procedure it was not clear how stations numbers relate to the eventual position of the station icon on the screen.

(b) It would be better to use object-oriented interaction. The user thus chooses the object to interact with and is then presented with a list of possible actions on that object.

The final model was developed to incorporate these comments.

A.6. Lessons learned.

The project provided some useful experience of animation prototyping:

(a) Timescales. The whole project took about 12 weeks. Much of that time was occupied with working out how to use a new version of the SIMSCRIPT II.5 package. With that knowledge now the same project could be accomplished much more quickly.

(b) Design approach. The decision to adopt an object-oriented problem decomposition was justified by the flexibility of the resulting model. The modifications made at each stage had a very limited effect on other parts of the program.

(c) Tool support. In retrospect use of a structured design tool such as PDF much earlier in the project would have also have been a great help. The specification was large and the model code increased in complexity very rapidly.

(d) Model demonstrations. Great care needs to be taken with what infromation is presented and the representation of information. Diagrams and symblos which are familiar to the client are ideal as the basis for displays. All information relavent to the state must be clearly displayed along with additional indications of what has happened and what is likely to happen next.

APPENDIX B.

B AN INTRODUCTION TO FORMAL SYSTEMS.

B.1. Introduction.

This chapter is not intended to be a tutorial discussing the inner working of formal systems. Rather, it aims to give information essential to the understanding of the concepts which are encountered. For those interested in a more detailed discussion of the subject the books by Reeves [Reeves90], Galton [Galton90], Denvir [Denvir86], Woodcock [Woodcock88] and Davis [Davis89] are recommended.

B.2. Formal Systems.

Formal systems consist of two parts, a formal language and a deductive apparatus. If a formal language is to be useful for reasoning about the world, the symbols in the language must be given some meaning, a semantics. This is achieved through the provision of an interpretation of the language.

B.2.1. Formal Languages.

A formal language consists of two parts - an alphabet and a syntax. An alphabet is all the symbols which are found in the language. A syntax is a set of rules specifying how these symbols may be combined together. An acceptable string of symbols in a language is called a well-formed formulae (wff). The formal language itself is thus just a collection of all its wff. Formal languages are often specified using a meta-language (a language for describing languages) such as Backus-Naur Form (BNF) or syntax diagrams.

B.2.2. Semantics - adding meaning to symbols.

The collections of symbols thus defined are meaningless; they bear no relationship to any quantities in the real world. To define the semantics of a formal language, meaning must be assigned to each wff allowed by the grammar. This is achieve by giving the language an interpretation. The interpretation assigns meaning from the real world (more properly called the "domain of interest") to

each wff. The exact details of interpretations is a very complex issue.

B.2.3. Inference systems.

The formal languages described thus far give the user descriptive power. The second part of a formal system is a deductive apparatus. This will give the user the ability to manipulate the symbols without regard to their meaning under any particular interpretation. This is in Jones' words [Jones90] "a game with symbols". The two components of a deductive apparatus are axioms and inference rules.

> Axioms (literally: self-evident truths) are the wffs which can be written down without reference to any other wffs in the language.
>
> Inference rules are rules which allow the production of wffs in the language as a direct consequence of other wffs.

B.2.4. Proofs and theorems.

A proof in a formal system is a finite sequence of wffs each of which is either a an axiom or an immediate consequence of one or more of the preceding wffs. A wff which can be proved within the formal system is said to be a theorem of the system. Note that all axioms are theorems of the system.

An interpretation of a formal system in which wffs denote statements which can be true or false is:-

> Consistent if every theorem of the system interprets to a true statement.
>
> Complete if every true statement can be proved as a theorem.

Unfortunately most useful formal systems are incomplete and there will thus be occasions when it will prove impossible to prove things which are known to be true. For an excellent introduction to the complex ideas of undecidability read Smullyan's book [Smullyan88] on Gödel and the limits of formalisation.

B.2.5. Derivations.

A derivation is the formal equivalent of the argument "if one is given that ... then one can deduce that ...". A derivation begins with a set of wffs, *P*, called premises. From this a sequence of wffs is formed such that each preceding wff is either an axiom, a premise or a direct consequence of one or

more of the preceding wffs. The last wff, W, in the sequence of wffs is called a derivation in the formal system. If there is a derivation from P to W one can write $P \vdash W$. The symbol $\vdash$ is known as a syntactic turnstile. It is a metasymbol, i.e. it is not part of the formal language itself, but allows statements about the formal system to be made. A derivation is simply a manipulation of the symbols of the wffs, there is no consideration of the meaning of those symbols.

It is important to note that a proof is a special type of derivation in which there are no premises. Similarly a derivation, $P \vdash W$, in a formal system F corresponds to a proof, $\vdash W$, in another richer system F' in which all premises, P, are incorporated as axioms along with the axioms and inference rules of F.

B.3. A Simple Example of a Formal System.

B.3.1. Propositions.

A proposition is an expression which can have the value true or false, but not both. The following statements, for example, are simple propositions:-

Bees are insects.
Some cats are black.
Loughborough is in Devon.

Simple propositions may be combined together, as described below, to form compound propositions. There are three basic laws of propositions:-

(a) Law of the excluded middle. A proposition must be either true or false.
(b) Contradiction. A proposition cannot be both true and false.
(c) Truth functionality. The truth value of a compound proposition is uniquely determined by the truth value of its constituent parts.

B.3.2. A formal language for propositions.

In order to represent propositions and combine them together, an alphabet and a syntax is needed. The

alphabet which will be considered here is as follows:-

$$\{P,Q,R,\ldots,P_1,P_2,\ldots,\wedge,\vee,\Rightarrow,\Leftrightarrow,\neg\}$$

Sentences which represent propositions and combinations of propositions can be formed using the following grammar rule:-

sentence = "P" | "Q" | "R" | ... | "P_1" | "Q_1" | ...
| "¬", *sentence*
| "(", *sentence*, " ∨ ", *sentence*, ")"
| "(", *sentence*, " ∧ ", *sentence*, ")"
| "(", *sentence*, "⇒", *sentence*, ")"
| "(", *sentence*, "⇔", *sentence*, ")";

The following are all sentences in this language:-

$(P \Leftrightarrow Q)$
$((P \wedge Q_1) \Rightarrow R)$
$(((P \vee Q) \vee (P \wedge R)) \Rightarrow \neg R)$

B.3.3. Semantics for propositions.

In this system the symbol P, Q, R etc. are the truth values of simple propositions. The symbols $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$ represent ways of combining simpler propositions to form compound propositions. These symbols are called connectives. Conventionally the connectives given above have names. The names of the connectives are:-

$\neg$ Not. The sentence $\neg P$ is referred to as the negation of P. The negation of a sentence can be thought of as the logical opposite of that sentence.

$\wedge$ And. The sentence $P \wedge Q$ is referred to the conjunction of P and Q. P and Q are conjuncts of the sentence.

$\vee$ Or. The sentence $P \vee Q$ is referred to as the disjunction of P and Q. P and Q are disjuncts of the sentence. The truth value of the sentence is "P or Q or both P and Q". It is also referred to as inclusive or.

$\Rightarrow$ Implication or "if ... then ... ". The sentence $P \Rightarrow Q$ is read as P implies Q. Care is needed in the informal interpretation of this connective as, in programming terms, there is no information covering "otherwise" or "else". Implication should always be referred to truth tables for exact interpretation.

$\Leftrightarrow$ Equals. This connective is also referred to as double implication. The sentence $P \Leftrightarrow Q$ is true if and only if P and Q have the same truth value.

The principle of truth functionality is adhered to and the truth values of compound expression are determined by the truth values of their constituent propositions. The connectives have a constant meaning in different interpretations, but the meanings of the propositions P, Q, R ... may vary. The meaning of compound forms can be given in a concise way using truth tables. In the following tables A, B are simple or compound expression. T and F denote the logical values true and false respectively. For simple sentences involving only one proposition the meaning is:-

A	(A)	$\neg A$
T	T	F
F	F	T

Note that it is necessary to give an interpretation to sentences of the form (A) as the same as A for use when interpreting compound propositions containing brackets.

For sentences which contain two component propositions and a connective the meaning is:-

A	B	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

The process of assigning a truth value to a sentence built from connectives and propositions consists

of:-

(a) giving an interpretation to the symbols denoting simple propositions.

(b) using the truth tables above to evaluate a truth value for the sentence.

The brackets in the expression are used to determine the order of evaluation of the compound expression, with inner brackets being evaluated first. In order to demonstrate this consider the expression $(\neg(P \wedge Q) \Leftrightarrow Q)$. Using a truth table to layout all possible interpretations of this sentence gives:-

P	Q	$(P \wedge Q)$	$\neg(P \wedge Q)$	$(\neg(P \wedge Q) \Leftrightarrow Q)$
T	T	T	F	F
T	T	F	T	F
F	T	F	T	T
F	F	F	T	F

This table shows the four possible interpretations of the sentence. If only certain interpretation were of interest all rows in the column would not need evaluation. The clear problem with such a mechanism is the very rapid expansion of the number of possible interpretations as the number of constituent propositions increases. This is the motivation for developing a propositional calculus whereby interpretations may be examined by syntactic manipulation. That is, sentences may be manipulated prior to any interpretation.

Sentences may be classified as follows:-

(a) Some sentences always interpret to true. Such a sentences is called a tautology or is said to be valid.

(b) Some sentences always interpret to false. Such sentences are called inconsistencies or contradictions and are said to be inconsistent.

(c) Finally some sentences are neither tautologies nor inconsistencies, their truth value depends on that of their constituent propositions. They are called contingencies.

(d) Tautologies and contingencies are said to be consistent as they evaluate to true under at least one interpretation.

If for some list of sentences P, whenever all the sentences of P are true, then some sentence W is true, then W is said to be a semantic consequence of the list of sentences P. This is written as $P \models W$. The symbol $\models$ is another metasymbol and is called the semantic turnstile. The statement $P \models W$ can be paraphrased as, given a set of assumptions P are true then W is also true. For example to demonstrate that P, $P \wedge Q \models Q$ the truth table is:-

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

Whenever the two sentences on the left of the expression are true (in the first row) the sentence on the right is true, thus P, $P \wedge Q \models Q$. It is interesting to compare semantic consequence and syntactic consequence. With the syntactic consequence, symbols are manipulated without reference to their possible meaning. Having considered the language of propositions and its semantics the next section considers a deductive apparatus for propositions.

B.3.4. Propositional calculus.

Propositional calculus is the formal deductive apparatus which facilitates reasoning about the truth value of simple and compound propositions. The purpose of a deductive apparatus is to allow reasoning to take place at a purely syntactic level. In this way propositions may be analysed before possible interpretations are made explicit (c.f. truth tables above). There are many possible choices of deductive apparatus [Reeves90, Galton90], in VDM, and hence here the system chosen is natural deduction. This system consists of a number of rules, called inference rules, for introducing and eliminating the basic connectives. These rules all have long names, e.g $\wedge$-Introduction and shortened names, e.g. $\neg$-E. The form of these rules is a list of sentences above a horizontal line and a sentence below the line. The meaning of this is that if all the sentences above the line are given then the sentence below the line can be deduced as an immediate consequence. The only explanation required is to cite the appropriate rule. This is done by writing the name of the rule and what sentences are taken as given.

The basic rules are set out below, where A and B are any simple or compound propositions:-

∧-Introduction (∧-I)

$$\frac{A,B}{A \wedge B} \quad \text{and} \quad \frac{A,B}{B \wedge A}$$

∧-Elimination (∧-E)

$$\frac{A \wedge B}{A} \quad \text{and} \quad \frac{A \wedge B}{B}$$

∨-Introduction (∨-I)

$$\frac{A}{A \vee B} \quad \text{and} \quad \frac{A}{B \vee A}$$

¬-Elimination (¬-E)

$$\frac{\neg \neg A}{A}$$

⇒-Elimination (⇒-E)

$$\frac{A, A \Rightarrow B}{B}$$

⇔-Elimination (⇔-E)

$$\frac{A \Leftrightarrow B}{A \Rightarrow B} \quad \text{and} \quad \frac{A \Leftrightarrow B}{B \Rightarrow A}$$

⇔-Introduction (⇔-I)

$$\frac{A \Rightarrow B, A \Rightarrow B}{A \Leftrightarrow B}$$

Performing derivations using this subset of the formal system is simply a matter of matching the grammatical form of the sentence under consideration to the appropriate rule for changing connectives.

Consider the following derivation:-

$P \wedge (Q \wedge R) \vdash (P \wedge Q) \wedge R$

from $P \wedge (Q \wedge R)$

1 P . . . . . $\wedge$-E(h)

2 $Q \wedge R$ . . . . . $\wedge$-E(h)

3 Q . . . . . $\wedge$-E(2)

4 R . . . . . $\wedge$-E(2)

5 $P \wedge Q$ . . . . . $\wedge$-I(1,3)

infer $(P \wedge Q) \wedge R$ . . . . . $\wedge$-I(4,5)

The layout of the derivation here needs some explanation as it will be used frequently. The first line, from ..., lists the premise (possibly more than one) of the derivation. The numbered lines are the steps in the proof with the right hand annotations giving the justification for each step. Justifications list the inference rule applied and the line number of the wff(s) to which it is applied. The final line, infer ... , is the desired conclusion.

The set of inference rules is not yet complete. The remaining rules are more complicated in that they make use of assumptions. Assumptions are internal to a proof and a used to make sub-derivations. When the desired result has been derived the assumption is discharged. The derivation *Assumption* $\vdash$ *Conclusion* may be used for further derivations but no formulae derived within the sub-derivation may be used outside that sub-derivation. The remaining three rules are:-

$\vee$-Elimination ($\vee$-E)

$$\frac{A \vdash C, B \vdash C, A \vee B}{C}$$ Where A and B are assumptions.

$\Rightarrow$-Introduction ($\Rightarrow$-I)

$$\frac{A \vdash B}{A \Rightarrow B}$$ Where A is an assumption.

¬-Introduction (¬-I)

$$\frac{A \vdash B, A \vdash \neg B}{\neg A}$$ Where A is an assumption.

The rule ∨-E is a formalised statement of the procedure of reasoning by cases. The rule ¬-I formalises the principle of proof by contradiction. These new rules are illustrated in the following derivation:-

$P \Rightarrow Q, Q \Rightarrow R \vdash P \Rightarrow R$

from $P \Rightarrow Q, Q \Rightarrow R$
Subderivation(1)
1 from P
1.1 Q . . . . . ⇒-E(h,h1)
infer R . . . . . ⇒-E(h,1.1)
infer $P \Rightarrow R$ . . . . . ⇒-I(1)

The main point illustrated here is the use of sub-derivation(1) to provide the sequent $P \vdash R$. This is then used in the ⇒-I rule to justify the final line of the derivation. The following derivation, which demonstrates the associativity of ∨, shows the nesting of sub-derivations and discharge of assumptions.

$(P \vee Q) \vee R \vdash P \vee (Q \vee R)$

from $(P \vee Q) \vee R$
sub-derivation(1)
1 from $P \vee Q$
sub-derivation(1.1)
1.1 from P
infer $P \vee (Q \vee R)$ . . . . . ∨-I(h1.1)
sub-derivation(1.2)
1.2 from Q
1.2.1 $Q \vee R$ . . . . . ∨-I(h1.2)
infer $P \vee (Q \vee R)$ . . . . . ∨-I(1.2.1)
infer $P \vee (Q \vee R)$ . . . . . ∨-E(h1,1.1,1.2)
sub-derivation(2)
2 from R

2.1 $Q \vee R$ . . . . . $\vee$-I(h2)

infer $P \vee (Q \vee R)$ . . . . . $\vee$-I(2.1)

infer $P \vee (Q \vee R)$ . . . . . $\vee$-E(h,1,2)

Finally there are two important properties of propositional calculus which make it a very useful formal system; it is consistent and complete. These properties are expressed as follows:-

(a) A formal system is complete with respect to an interpretation if for a set of sentences P whenever $P \models W$ then $P \vdash W$. This means that if interpretations of P and W can be reasoned about formally then that reasoning can also be conducted formally.

(b) A formal system is consistent with respect to an interpretation if for a set of sentences P whenever $P \vdash W$ then $P \models W$. This means that if something can be derived formally then reasoning about the meanings of the sentences concerned will arrive at the same conclusion.

In propositional calculus these properties mean that if the derivation $P \vdash W$ can be performed then $P \models W$. Hence if all the sentences of P are given interpretations of true then W also interprets to true. Thus syntactic manipulations in the calculus can be used to deduce true things from other true things.

The simple formal system of propositional calculus, introduced above, is useful for reasoning about the truth of simple or compound propositions. In the next section the more powerful formal system of predicate calculus is introduced.

B.4. A More Powerful Formal System.

B.4.1. Predicates.

The simple propositional calculus already described can be extended with a small number of extra constructs to allow reasoning about objects and properties of objects. More specifically predicates are used to capture properties of objects and relationships between objects. The components of the formal language which map on to the properties of objects are called predicates. The name given to predicates is conventionally related to the property captured by the predicate e.g. *student*(x) might be a predicate which captures the property of being a student. The x is a free variable, which when filled by the name of a suitable object to create a proposition. For example, *student*(*Tim*) will be true if the object denoted by *Tim* is a student and false if *Tim* is not. Predicates may express relationships between more

than one object. The predicate *married(x,y)* could be used to capture the idea that *x* is married to *y*. The names of predicates are not in themselves important, interpretations will ultimately place meaning on the predicates.

B.4.2. Predicate logic.

Extending the formal language for propositions to allow the expression predicates can be done using the following definition.

sentence = *simple_proposition* | *predicate*
| "¬", *sentence*
| "(", *sentence*, " ∨ ", *sentence*, ")"
| "(", *sentence*, " ∧ ", *sentence*, ")"
| "(", *sentence*, "⇒", *sentence*, ")"
| "(", *sentence*, "⇔", *sentence*, ")"
| "∀", *variable_name*, " • ", *sentence*;
| "∃", *variable_name*, " • ", *sentence*;
simple_proposition = "*P*"|"*Q*"|"*R*"|...;
predicate = *predicate_name*, "(", *termlist*, ")";
predicate_name = ... (* Any arbitrary string of characters *);
termlist = *term* | *term*, ",", *termlist*;
term = *proper_name* | *arbitrary_name* | *variable_name*;
proper_name = ... (* Any arbitrary string of characters *);
arbitrary_name = "*a*"|"*b*"|"*c*"|...;
variable_name = "*x*"|"*y*"|"*z*"|...;

The two new symbols ∃ and ∀ are used to capture the ideas of universal and existential quantification. Universal quantification is used to express propositions of the form "every object has this property" or "all objects are related in this way". Existential quantification is used to express propositions of the form "there is at least one object which has this property".

In this language one may say:-

$\forall x \cdot P(x)$
$\exists x \cdot \forall y \cdot Q(x,y)$

Predicates themselves have no truth value. Predicates can give rise to propositions in two ways:-

(a) by instantiating their free variables with the names of objects.

(b) by describing the instantiation process itself using the technique of quantification. This is done using the two new symbols.

For a predicate $P(x)$, with free variable x, both the sentences

$$\exists x \cdot P(x) \quad \forall x \cdot P(x)$$

are propositions, called quantified expressions. The quantification binds the variable, it is said that x is bound by the quantification, so that it can no longer be instantiated.

B.4.3. A semantics for predicate logic.

The conventional interpretation for predicate logic is similar, but more complex, than that for propositional logic. Any simple propositions and the connectives are interpreted as in propositional logic. Proper names used are related to objects in the real world. Bound variables in quantified expressions are not assigned meanings. Predicates denote properties of objects or relationships between objects. The number of free variables in the predicate should be carefully chosen to match the idea it expresses.

The validity of sentences in predicate logic can be examined semantically. However this is much more difficult than for propositional logic. With universal quantification of very large domains of interest the properties of many objects will need examining. Similarly with existential quantification finding a single object for which the property holds may be a very long task. Truth tables will not work with predicate logic in general and unfortunately there is no other similar mechanism which can show general validity. Fortunately though there is a deductive apparatus for predicate calculus which is both consistent and complete for the class of interpretations outlined above.

B.4.4. Predicate calculus.

Predicate calculus is the formal deductive apparatus which facilitates reasoning about the truth value of simple and compound predicates. Following the above trends the deductive apparatus for predicates is an extension of that for propositional calculus. As there are two new symbols which have been introduced there are four new corresponding inference rules. These are, not surprisingly:-

∀-Elimination (∀-E)

$$\frac{\forall x \cdot P(x)}{P(a)}$$ where a is arbitrary.

∀-Introduction (∀-I)

$$\frac{P(a)}{\forall x \cdot P(x)}$$ where a is arbitrary.

∃-Elimination (∃-E)

$$\frac{\exists x \cdot P(x)}{P(q)}$$ where q is a particular name.

∃-Introduction (∃-I)

$$\frac{P(a)}{\exists x \cdot P(x)}$$ where t is any term.

These inference rules may be used in the same fashion as those for propositional calculus. As the deductive apparatus is consistent and complete, the results for syntactic and semantic consequences being equivalent hold. Therefore the predicate calculus may be used to analyse the truth of sentences in the predicate logic.

B.5. Summary.

In this appendix the basic terminology of formal systems has been introduced. The concepts of formal languages and semantics have been shown and their application to the formal systems of propositional logic and predicate logic has been shown. The power of such systems come from their deductive apparatus, which allows the analysis of complex sentences to determine their truth. The use of the propositional calculus in derivation has been shown.

Finally the reader should be aware that the terms propositional calculus and predicate calculus are widely used. The systems actually referred to often differ widely or subtly from those detailed here. The assumptions such systems are based on, as well as the interpretations used, must be carefully examined. Formal systems for software engineers, like tools for other engineers, should only be used carefully and in accordance with the manufacturer's instructions.

APPENDIX C.

C MATHEMATICAL DETAILS OF VDM.

C.1. Defining Data Types.

C.1.1. Simple data types.

VDM specifications are built by modelling a system in terms of basic mathematical entities. These entities are:-

(a) Boolean variables, here written as **B**.
(b) Integer numbers, here written as **Z**.
(c) Natural numbers, positive integers (including zero), here written as **N**.
(d) Natural numbers, strictly positive integers (excluding zero), here written as $\mathbf{N_1}$.
(e) Real numbers, here written as **R**.

In addition to these VDM includes three further entities. These are:-

(a) Finite Sets. A set is an unordered collection of distinct objects.
(b) Finite Maps. Maps are associations between two sets of elements, the key set and the value set. Each, unique, key element is associated with an element in the value set.
(c) Finite Sequences. A sequence is an ordered collection of objects.

Specific instances of the above three entities type may be defined in two ways, enumeration and comprehension:-

For sets:

enumeration:

$s = \{ 1, 2, 4 \}$

$primary\_colours = \{$ RED, GREEN, BLUE $\}$

comprehension:

$numbers = \{ n \in \mathbf{N} \mid 1 \leq n \leq 5 \}$

where $\in$ denotes set membership

i.e. $numbers = \{ 1, 2, 3, 4, 5 \}$

For maps:

enumeration:

$m = \{1 \mapsto x_1, 3 \mapsto x_4, 6 \mapsto x_1, 2 \mapsto x_3, 10 \mapsto x_4\}$

directions = {11 ↦ UP, 22 ↦ LEFT, 33 ↦ DOWN, 44 ↦ RIGHT}

comprehension:

$squares = \{ x \mapsto x^2 \in \mathbf{N} \times \mathbf{N} \mid x \in \{i \in \mathbf{N} \mid -2 \leq i \leq 3 \} \}$

i.e. $square = \{ -2 \mapsto 4, -1 \mapsto 1, 0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9 \}$

For sequences:

enumeration:

queue = [120, 41, 1034, 4593]

comprehension:

$doubles = [x \in numbers \mid doubles(x) = 2 \times x]$

i.e. *doubles* = [2, 4, 6, 8, 10]

or as previously for a sequence of natural numbers:

$stack = \mathbf{N}^*$

These simple entities may be used to describe simple data types to be used in a specification.

C.1.2. Composite objects as data types.

These simple entities may be used to describe simple data types to be used in a specification. Further to this VDM has a mechanism for combining simpler entities together to form composite objects. In many respects these composite objects are like Pascal records. Composite objects have a number of fields, of specified types, and tags (names) for each of those fields. A composite type for a buffer of natural numbers of defined size may be defined as:-

compose *Buffer* of

$Max\text{-}size : \mathbf{N}_1$,

$Store : \mathbf{N}^*$

end

To create an object of this type and assign values to the appropriate field the concept of make-functions

is used. To create an empty buffer of size 256 numbers:-

$$mk\text{-}Buffer(\ 256,\ [])$$

The values of individual fields may be refered to using selector functions. Selectors when applied to composite values yield the component value. In the above example of a buffer the are two selector functions. Their signatures are:-

$$Max\text{-}size : Buffer \rightarrow \mathbf{N}_1$$
$$Store : Buffer \rightarrow \mathbf{N}^*$$

To change individual fields in a composite object there are a family of μ-functions. Consider the following composite object:

$$Buf = mk\text{-}Buffer(\ 256,\ [])$$

To change the size of the buffer the function is:

$$\mu(\ Buf,\ Max\text{-}size \mapsto 128\ )$$

The resulting object is equivalent to the object formed by writing:-

$$mk\text{-}Buffer(\ 128,\ [])$$

The selector functions can be used on this object:

$$Max\text{-}size(Buf) = 128$$
$$Store(Buf) = []$$

To change the store field the function is:-

$$\mu(\ Buf,\ Store \mapsto [\ 200,\ 6,\ 6300\ ]$$

Composite objects are most frequently used in the specifications of state variables and so names may be associated with the set of composite objects defined:-

Finite_Buffer = compose *Buffer* of

Max-size : $\mathbf{N}_1$

Store : $\mathbf{N}_*$

end

The definition of *Buffer* is not yet complete as the size of the *Store* field has not been constrained. It is necessary to restrict the possible combinations of values taken by this data type. This is achieved by the use of a "data type invariant". This is a truth-valued function which is true for all valid object of the type. Invariants are defined as follows:-

$$inv\text{-}Buffer(Buf) \triangleq \text{len } Store(Buf) \leq Max\text{-}size(Buf)$$

This says that for valid objects of type *Buffer* the number of elements in the *Store* sequence is always less than or equal to the *Max-size* of the buffer.

C.2. An example implementability proof.

Brielfy, a satisfiability or implementability proof is a formal demonstration that for all valid inputs and initial states there exists some outputs and final states which are valid. The formal statement of this is:-

if σ is the state variable and Σ is its type

$$\forall \underline{\sigma} \in \Sigma \cdot pre\text{-}OP(\underline{\sigma}) \Rightarrow \exists \sigma \in \Sigma \cdot post\text{-}OP(\underline{\sigma}, \sigma)$$

For the *Add_item* operation above this becomes:-

$$\forall \underline{Buf} \in Buffer,\ item \in \mathbf{N} \cdot \text{true} \Rightarrow \exists Buf \in Buffer \cdot$$
$$(\ \text{len } \underline{Store(Buf)} < Max\text{-}Size(Buf) \wedge$$
$$Store(Buf) = \underline{Store(Buf)} \frown [item]\ ) \vee$$
$$(\ \text{len } \underline{Store(Buf)} = Max\text{-}Size(Buf) \wedge$$
$$Store(Buf) = \text{tail}(\underline{Store(Buf)}) \frown [item]\ )$$

As *Buf* is a composite object this can be rewritten in terms of its constituent fields as:-

$$\forall\ \underline{Store(Buf)} \in \mathbf{N}^*, Max\text{-}size(Buf) \in \mathbf{N}_1,\ item \in \mathbf{N} \bullet$$
$$\text{true} \Rightarrow \exists Store(Buf) \in \mathbf{N}^* \bullet$$
$$((\ \text{len}\ \underline{Store(Buf)} < Max\text{-}size(Buf)\ \wedge$$
$$Store(Buf) = \underline{Store(Buf)} \frown [item]\ )\ \vee$$
$$(\ \text{len}\ \underline{Store(Buf)} = Max\text{-}size(Buf)\ \wedge$$
$$Store(Buf) = \text{tail}(\underline{Store(Buf)}) \frown [item]\ )\ )\ \wedge$$
$$inv\text{-}Buffer(Buf)$$

Note also the introduction of the data type invariant. Any final values of the state must be valid objects of the appropriate type. Using the distribution of $\wedge$ over $\vee$ and expanding the invariant this is rewritten as :-

$$\forall\ \underline{Store(Buf)} \in \mathbf{N}^*, Max\text{-}size(Buf) \in \mathbf{N}_1,\ item \in \mathbf{N} \bullet$$
$$\text{true} \Rightarrow \exists Store(Buf) \in \mathbf{N}^* \bullet$$
$$(\ \text{len}\ \underline{Store(Buf)} < Max\text{-}size(Buf)\ \wedge$$
$$Store(Buf) = \underline{Store(Buf)} \frown [item]\ \wedge$$
$$\text{len}\ Store(Buf) \leq Max\text{-}size(Buf)\ )\ \vee$$
$$(\ \text{len}\ \underline{Store(Buf)} = Max\text{-}size(Buf)\ \wedge$$
$$Store(Buf) = \text{tail}(\underline{Store(Buf)}) \frown [item]\ \wedge$$
$$\text{len}\ Store(Buf) \leq Max\text{-}size(Buf)\ )$$

The proof will take the form of a sequent:-

$$\text{from}\ \underline{Store(Buf)} \in \mathbf{N}^*, Max\text{-}size(Buf) \in \mathbf{N}_1,\ item \in \mathbf{N} \vdash$$
$$\text{true} \Rightarrow \exists Store(Buf) \in \mathbf{N}^* \bullet$$
$$(\ \text{len}\ \underline{Store(Buf)} < Max\text{-}size(Buf)\ \wedge$$
$$Store(Buf) = \underline{Store(Buf)} \frown [item]\ \wedge$$
$$\text{len}\ Store(Buf) \leq Max\text{-}size(Buf)\ )\ \vee$$
$$(\ \text{len}\ \underline{Store(Buf)} = Max\text{-}size(Buf)\ \wedge$$
$$Store(Buf) = \text{tail}(\underline{Store(Buf)}) \frown [item]\ \wedge$$
$$\text{len}\ Store(Buf) \leq Max\text{-}size(Buf)\ )$$

Sub-derivation(1)

1 from $\underline{Store(Buf)} \in \mathbf{N}^*$, $Max\text{-}size(Buf) \in \mathbf{N}_1$, $item \in \mathbf{N}$

Subderivation(1.1)

1.1 from $B \in Buffer$

1.1.1 len *Store(B)* ≤ *Max-size(B)* . . . . . *inv-Buffer*(h1.1)

infer len *Store(B)* < *Max-size(B)* ∨

Store(B) = *Max-size(B)* . . . . . property-≤(1.1.1)

Subderivation(1.2)

1.2 from len *Store(Buf)* < *Max-size(Buf)*

Subderivation(1.2.1)

1.2.1 from *Store(Buf)* = *Store(Buf)* ⌢ [*item*]

1.2.1.1 len *Store(Buf)* < *Max-size(Buf)* ∧

Store(Buf) = *Store(Buf)* ⌢ [*item*] . . . . . ∧-I(h1.2, h1.2.1)

1.2.1.2 len *Store(Buf)* ⌢ [*item*] ≤

Max-size(Buf) . . . . . prop-⌢(h1.2)

1.2.1.3 len *Store(Buf)* ≤

Max-size(Buf) . . . . . ≡-subs(1.2.1.2,h1.2.1)

1.2.1.4 len *Store(Buf)* < *Max-size(Buf)* ∧

Store(Buf) = *Store(Buf)* ⌢ [*item*] ∧

len *Store(Buf)* ≤

Max-size(Buf) . . . . . ∧-I(1.2.1.1, 1.2.1.4)

infer (len *Store(Buf)* < *Max-size(Buf)* ∧

Store(Buf) = *Store(Buf)* ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) ∨

(len *Store(Buf)* = *Max-size(Buf)* ∧

Store(Buf) = tail(*Store(Buf)*) ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) . . . . . ∨-I(1.2.1.4)

Subderivation(1.2.2)

1.2.2 from *Store(Buf)* = *Store(Buf)* ⌢ [*item*]

1.2.2.1 [*item*] ∈ $\mathbf{N}^*$ h1

1.2.2.2 *Store(Buf)* ∈ $\mathbf{N}^*$ . . . . . defn-*Buffer*(h1)

1.2.2.3 *Store(Buf)* ⌢ [*item*] ∈ $\mathbf{N}^*$ . . . . . ⌢-defn(1.2.2.1, 1.2.2.2)

infer *Store(Buf)* ∈ $\mathbf{N}^*$ . . . . . ≡-subs(h1.2.2, 1.2.2.3)

infer ∃ *Store(Buf)* ∈ $\mathbf{N}^*$ •

(len *Store(Buf)* < *Max-size(Buf)* ∧

Store(Buf) = *Store(Buf)* ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) ∨

(len *Store(Buf)* = *Max-size(Buf)* ∧

Store(Buf) = tail(*Store(Buf)*) ⌢ [*item*] ∧

len *Store(Buf)* ≤

Max-size(Buf)) . . . . . ∃-I(1.2.1, 1.2.2)

Subderivation(1.3)

1.3 from len *Store(Buf)* = *Max-size(Buf)*

Subderivation(1.3.1)

1.3.1 from *Store(Buf)* = tail(*Store(Buf)*) ⌢ [*item*]

1.3.1.1 len *Store(Buf)* = *Max-size(Buf)* ∧

Store(Buf) =

tail *Store(Buf)* ⌢ [*item*] . . . . . ∧-I(h1.3, h1.3.1)

1.3.1.2 len tail (*Store(Buf)*) =

Max-size(Buf)-1 . . . . . prop-tail(h1.2)

1.3.1.3 len (tail (*Store(Buf)*) ⌢ [*item*] =

Max-size(Buf) . . . . . len-i(h1.2)

1.3.1.4 len *Store(Buf)* = *Max-Size(Buf)* . . . . . ≡-subs(h,1.2.2.3)

1.3.1.5 len *Store(Buf)* ≤ *Max-Size(Buf)* . . . . . prop-≤(h,1.2.2.3)

1.3.1.6 (len *Store(Buf)* = *Max-size(Buf)* ∧

Store(Buf) = tail(*Store(Buf)*) ⌢ [*item*] ∧

len *Store(Buf)* ≤

Max-size(Buf)) . . . . . ∧-I(1.3.1.1, 1.3.1.6)

infer (len *Store(Buf)* < *Max-size(Buf)* ∧

Store(Buf) = *Store(Buf)* ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) ∨

(len *Store(Buf)* = *Max-size(Buf)* ∧

Store(Buf) = tail(*Store(Buf)*) ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) . . . . . ∨-I(1.3.1.6)

Subderivation(1.3.2)

1.3.2 from *Store(Buf)* = tail(*Store(Buf)*) ⌢ [*item*]

1.3.2.1 [*item*] ∈ N^* h1

1.3.2.2 *Store(Buf)* ∈ N^* . . . . . defn-*Buffer*(h1)

1.3.2.3 tail (*Store(Buf)*) ∈ N^* . . . . . prop-tail(1.3.2.2)

1.3.2.4 tail(*Store(Buf)*) ⌢

[*item*] ∈ N^* . . . . . prop-⌢(1.3.2.4, 1.3.2.1)

infer *Store(Buf)* ∈ N^* . . . . . ≔-subs(h1.3.2, 1.3.2.4)

infer ∃ *Store(Buf)* ∈ N^* •

(len *Store(Buf)* < *Max-size(Buf)* ∧

Store(Buf) = *Store(Buf)* ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) ∨

(len *Store(Buf)* = *Max-size(Buf)* ∧

Store(Buf) = tail(*Store(Buf)*) ⌢ [*item*] ∧

len *Store(Buf)* ≤

Max-size(Buf)) . . . . . ∃-I(1.3.1,1.3.2)

1.4 ∃ *Store(Buf)* ∈ N^* •

(len *Store(Buf)* < *Max-size(Buf)* ∧

Store(Buf) = *Store(Buf)* ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) ∨

(len *Store(Buf)* = *Max-size(Buf)* ∧

Store(Buf) = tail(*Store(Buf)*) ⌢ [*item*] ∧

len *Store(Buf)* ≤

Max-size(Buf)) . . . . . ∨-E(1.1,1.2,1.3)

infer true ⇒ ∃*Store(Buf)*∈N^* •

(len *Store(Buf)* < *Max-size(Buf)* ∧

Store(Buf) = *Store(Buf)* ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) ∨

(len *Store(Buf)* = *Max-size(Buf)* ∧

Store(Buf) = tail(*Store(Buf)*) ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) . . . . . vac⇒-I(1.4)

infer ∀ *Store(Buf)*∈N^*, *Max-size(Buf)*∈N_1, *item*∈N •

true ⇒ ∃*Store(Buf)*∈N^* •

(len *Store(Buf)* < *Max-size(Buf)* ∧

Store(Buf) = *Store(Buf)* ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) ∨

(len *Store(Buf)* = *Max-size(Buf)* ∧

Store(Buf) = tail(*Store(Buf)*) ⌢ [*item*] ∧

len *Store(Buf)* ≤ *Max-size(Buf)*) . . . . . ∀-I(1)

This proof can be explained as follows:-

(a) The main sub-derivation (1) is to show that:-

$$
\begin{aligned}
&\text{true} \rightarrow \exists \mathit{Store(Buf)} \in \mathbf{N}^* \bullet \\
&(\ \text{len } \underline{\mathit{Store(Buf)}} < \mathit{Max\text{-}size(Buf)} \wedge \\
&\quad \mathit{Store(Buf)} = \underline{\mathit{Store(Buf)}} \frown [\mathit{item}] \wedge \\
&\quad \text{len } \mathit{Store(Buf)} \leq \mathit{Max\text{-}size(Buf)}\ ) \vee \\
&(\ \text{len } \underline{\mathit{Store(Buf)}} = \mathit{Max\text{-}size(Buf)} \wedge \\
&\quad \mathit{Store(Buf)} = \text{tail}(\underline{\mathit{Store(Buf)}}) \frown [\mathit{item}] \wedge \\
&\text{len } \mathit{Store(Buf)} \leq \mathit{Max\text{-}size(Buf)}\ )
\end{aligned}
$$

is a logical consequence of the assumption that:-

$$\underline{\mathit{Store(Buf)}} \in \mathbf{N}^*, \mathit{Max\text{-}size(Buf)} \in \mathbf{N}_1,\ \mathit{item} \in \mathbf{N}$$

this is stated in sequent form in the first line of the proof.

(b) Sub-derivation(1.1) shows the property of the invariant that if the length of *Store* is less than or equal to *Max-size* it is either less than the *Max-size* or it is equal to the *Max-size*. This is obvious, but the technique used to examine the post-condition is argument by cases. Here it is established that there are two cases to argue.

(c) Accordingly, sub-derivation(1.2) argues the first case. Here the aim is to show that a solution to the post-condition exists if the buffer has at least one space left in it. Formally this is done by showing that:-

$$
\begin{aligned}
&\exists\ \mathit{Store(Buf)} \in \mathbf{N}^* \bullet \\
&\quad (\ \text{len } \underline{\mathit{Store(Buf)}} < \mathit{Max\text{-}size(Buf)} \wedge \\
&\quad\ \mathit{Store(Buf)} = \underline{\mathit{Store(Buf)}} \frown [\mathit{item}] \wedge \\
&\quad\ \text{len } \mathit{Store(Buf)} \leq \mathit{Max\text{-}size(Buf)}\ ) \vee \\
&\quad (\ \text{len } \underline{\mathit{Store(Buf)}} = \mathit{Max\text{-}size(Buf)} \wedge \\
&\quad\ \mathit{Store(Buf)} = \text{tail}(\underline{\mathit{Store(Buf)}}) \frown [\mathit{item}] \wedge \\
&\quad\ \text{len } \mathit{Store(Buf)} \leq \mathit{Max\text{-}size(Buf)}\ )
\end{aligned}
$$

is a logical consequence of the assumption:-

len $\underline{Store(Buf)}$ < *Max-size(Buf)*

(d) There are two main parts to this sub-derivation; the further sub-derivations(1.2.1) and (1.2.2). These show that the logical expression is true under the assumptions made and that the value of *Store(Buf)* needed for the solution is of the correct type.

(e) Combining sub-derivations (1.2.1) and (1.2.2) leads to the conclusion that under the assumption of (1.2) a solution does indeed exist.

(f) Sub-derivation (1.3) now argues the second case. The style of rgument is almost identical to that for (1.2) and the conclusion is the same.

(g) Using the result of (1.1), (1.2) and (1.3), (1.4) deduces that it is generally true that there is a solution for the post-condition. This allows the discharge of the assumption made for sub-derivation(1).

(h) Finally, the conclusion is drawn that the specification of the operation is indeed implementable.

C.3. A Simple Plant Controller.

C.3.1. Informal specification of the system.

The following informal specification of a plant controller will form the basis for examples on operation decomposition:-

(a) The system controls three valves: inlet, outlet and vent.
(b) Each valve may be either open or closed.
(c) The fail-safe condition is inlet and outlet valves closed and the vent valve open.
(d) The plant always begins in the fail-safe condition and must end in the same condition.
(e) On demand for service the controller must perform the following sequence of actions:-
    (i) close the vent valve.
    (ii) open the inlet valve.

(iii) open the outlet valve.

(f) On demand for shut down the controller must perform the following sequence of events:-

(i) close the inlet valve.

(ii) close the outlet valve.

(iii) open the vent valve.

C.3.2. Formal specification of the system state.

In order to specify this using VDM the state is specified as follows:-

valve = { OPEN, CLOSED }

state *Plant* of

inlet : *valve*

outlet : *valve*

vent : *valve*

init $Plant_0 \triangleq$ *mk-Plant*(CLOSED, CLOSED, OPEN)

C.3.3. Formal specification of the plant controller.

The plant controller may be specified as a single operation, thus:-

Plant_Controller()

ext wr *inlet* : *valve*

wr *outlet* : *valve*

wr *vent* : *valve*

pre *inlet*(*Plant*) = CLOSED $\wedge$

outlet(*Plant*) = CLOSED $\wedge$

vent(*Plant*) = OPEN

post *inlet*(*Plant*) = CLOSED $\wedge$

outlet(*Plant*) = CLOSED $\wedge$

vent(*Plant*) = OPEN

C.3.4. Decomposing the plant controller operation.

When reasoning about operations and their effects the notion of triples is a useful one. A triple is of the form $\{P\}Q\{R\}$. Where P and R are truth valued expressions and Q is some operation. This notation asserts that if the state satisfies expression P then the application of operation Q will yield a state which satisfies the expression R. This is most commonly encountered in VDM in the form:-

$$\{pre\}S\{post\}$$

Using this shorthand notation the inference rule for sequential composition, called ;-*I*, is formally stated as:

;-*I*

$$\frac{\{pre_1\}S_1\{pre_2 \wedge post_1\};\{pre_2\}S_2\{post_2\}}{\{pre_1\}S_1;S_2\{post_1 \mid post_2\}}$$

where:-

$post_1 \mid post_2 \triangleq \exists\, \sigma_i \in \Sigma \cdot post_1\,(\underline{\sigma}, \sigma_i) \wedge post_2\,(\sigma_i, \sigma)$

this is called the composition of the post-conditions.

This rule may be used to show that sequential decompositon:-

Plant_Controller: *Service*; *Shutdown*

is consistent where:-

Service()

ext wr *inlet* : *valve*
wr *outlet* : *valve*
wr *vent* : *valve*

pre *inlet*(*Plant*) = CLOSED ∧
outlet(*Plant*) = CLOSED ∧
vent(*Plant*) = OPEN

post *inlet*(*Plant*) = OPEN ∧
outlet(*Plant*) = OPEN ∧
vent(*Plant*) = CLOSED

Shutdown()

ext wr *inlet* : *valve*
wr *outlet* : *valve*
wr *vent* : *valve*

pre *inlet*(*Plant*) = OPEN ∧
outlet(*Plant*) = OPEN ∧
vent(*Plant*) = CLOSED

post *inlet*(*Plant*) = CLOSED ∧
outlet(*Plant*) = CLOSED ∧
vent(*Plant*) = OPEN

In the above example the aim is to show that:-

{*pre-Plant_controller*} (*Service*; *Shutdown*) {*post-Plant_controller*}

the following results are the basis of the proof:-

pre-Plant_controller ⇔ *pre-Service*
pre-Shutdown ⇔ *post-Service*
post-Service | *post-Shutdown*
⇔ ∃ $inlet_i$, $outlet_i$, $vent_i$ •
post-Service((*inlet*(*Plant*) = CLOSED ∧
outlet(*Plant*) = CLOSED ∧
vent(*Plant*) = OPEN)) ,
($inlet_i$(*Plant*) = OPEN ∧
$outlet_i$(*Plant*) = OPEN ∧
$vent_i$(*Plant*) = CLOSED)) ∧
post-Shutdown(($inlet_i$(*Plant*) = OPEN ∧
$outlet_i$(*Plant*) = OPEN ∧
$vent_i$(*Plant*) = CLOSED),

$$(inlet(Plant) = \text{CLOSED} \wedge$$
$$outlet(Plant) = \text{CLOSED} \wedge$$
$$vent(Plant) = \text{OPEN}))$$
$$\Rightarrow post\text{-}Plant\_controller$$

C.3.5. Decomposing the Service operation.

Consider a decomposition of the operation *Service*. The decomposition will be a sequential composition of three operation each changing the state of one valve. The decomposition is:-

Service : *Close_vent*; *Open_inlet*; *Open_outlet*

The operations are defined as:-

Close_vent ()
ext wr *vent*(*Plant*) : *valve*
pre *vent*(*Plant*) = OPEN
post *vent*(*Plant*) = CLOSED

Open_inlet ()
ext wr *inlet*(*Plant*) : *valve*
pre *inlet*(*Plant*) = CLOSED
post *inlet*(*Plant*) = OPEN

Open_outlet ()
ext wr *outlet*(*Plant*) : *valve*
pre *outlet*(*Plant*) = CLOSED
post *outlet*(*Plant*) = OPEN

The proof that this design step is valid follows from the inference rule:-

weaken

$$\frac{pre_s \Rightarrow pre; \{pre\}S\{post\}; post \Rightarrow post_w}{\{pre_s\}S\{post_w\}}$$

This can be paraphrased as stating that an operation which satisfies a specification necessarily satisfies

a weaker one. A "weaker" specification is one with a narrower pre-condition or a wider post-condition.

The extended form of the sequential composition inference rule for three operations in sequence is:-

$$\frac{\{pre_1\}S_1\{pre_2 \wedge post_1\};\ \{pre_2\}S_2\{pre_3 \wedge post_2\};\ \{pre_3\}S_3\{post_3\}}{\{pre_1\}(S_1;S_2;S_3)\{post_1 \mid post_2 \mid post_3\}}$$

$$post_1 \mid post_2 \mid post_3 \triangleq \exists\, \sigma_i,\sigma_j \in \Sigma \bullet post_1\,(\underline{\sigma}, \sigma_i) \wedge post_2\,(\sigma_i, \sigma_j) \wedge post_2\,(\sigma_j, \sigma)$$

The proof of the above decomposition is as follows:-

$$\begin{aligned} \textit{pre-Service} \Leftrightarrow\ & \textit{inlet}(\textit{Plant}) = \text{CLOSED} \wedge \\ & \textit{outlet}(\textit{Plant}) = \text{CLOSED} \wedge \\ & \textit{vent}(\textit{Plant}) = \text{OPEN} \\ \Rightarrow\ & \textit{vent}(\textit{Plant}) = \text{OPEN} \\ \Rightarrow\ & \textit{pre-Close\_vent} \end{aligned}$$

$$\begin{aligned} \textit{post-Close\_vent} \Leftrightarrow\ & \textit{vent}(\textit{Plant}) = \text{CLOSED} \wedge \\ & \textit{inlet}(\textit{Plant}) = \underline{\textit{inlet}(\textit{Plant})} \wedge \\ & \textit{outlet}(\textit{Plant}) = \underline{\textit{outlet}(\textit{Plant})} \\ \Rightarrow\ & \textit{inlet}(\textit{Plant}) = \text{CLOSED} \\ \Rightarrow\ & \textit{pre-Open\_inlet} \end{aligned}$$

Note here the explicit statement of the fact that *Close_vent* does not affect the value of the state variables *inlet* and *outlet*. This comes from the external clause of the definition of *Close_vent* which states that the operation only affects the state of *vent(Plant)*.

$$\begin{aligned} \textit{post-Open\_inlet} \Leftrightarrow\ & \textit{inlet}(\textit{Plant}) = \text{OPEN} \wedge \\ & \textit{vent}(\textit{Plant}) = \underline{\textit{vent}(\textit{Plant})} \wedge \\ & \textit{outlet}(\textit{Plant}) = \underline{\textit{outlet}(\textit{Plant})} \\ \Rightarrow\ & \textit{outlet}(\textit{Plant}) = \text{CLOSED} \\ \Rightarrow\ & \textit{pre-Open\_outlet} \end{aligned}$$

$$
\begin{aligned}
&\textit{post-Close\_vent} \mid \textit{post Open\_inlet} \mid \textit{post Open\_outlet} \\
&\quad \Leftrightarrow \exists\ \textit{inlet}_i, \textit{vent}_i, \\
&\qquad \textit{inlet}_j, \textit{outlet}_j, \textit{vent}_j \bullet \\
&\qquad\quad \textit{post-Close\_vent} \\
&\qquad\quad ((\underline{\textit{inlet(Plant)}} = \text{CLOSED} \wedge \\
&\qquad\qquad \underline{\textit{outlet(Plant)}} = \text{CLOSED} \wedge \\
&\qquad\qquad \underline{\textit{vent(Plant)}} = \text{OPEN}), \\
&\qquad\quad (\textit{inlet}_i(\textit{Plant}) = \text{CLOSED} \wedge \\
&\qquad\qquad \textit{vent}_i(\textit{Plant}) = \text{CLOSED} \wedge \\
&\qquad\qquad \textit{outlet}_i(\textit{Plant}) = \text{CLOSED})) \wedge \\
&\qquad\quad \textit{post-Open\_inlet} \\
&\qquad\quad ((\textit{inlet}_i(\textit{Plant}) = \text{CLOSED} \wedge \\
&\qquad\qquad \textit{vent}_i(\textit{Plant}) = \text{CLOSED} \wedge \\
&\qquad\qquad \textit{outlet}_i(\textit{Plant}) = \text{CLOSED}), \\
&\qquad\quad (\textit{inlet}_j(\textit{Plant}) = \text{OPEN} \wedge \\
&\qquad\qquad \textit{outlet}_j(\textit{Plant}) = \text{CLOSED} \wedge \\
&\qquad\qquad \textit{vent}_j(\textit{Plant}) = \text{CLOSED})) \wedge \\
&\qquad\quad \textit{post-Open\_outlet} \\
&\qquad\quad ((\textit{inlet}_j(\textit{Plant}) = \text{OPEN} \wedge \\
&\qquad\qquad \textit{outlet}_j(\textit{Plant}) = \text{CLOSED} \wedge \\
&\qquad\qquad \textit{vent}_j(\textit{Plant}) = \text{CLOSED}), \\
&\qquad\quad (\textit{inlet}(\textit{Plant}) = \text{OPEN} \wedge \\
&\qquad\qquad \textit{outlet}(\textit{Plant}) = \text{OPEN} \wedge \\
&\qquad\qquad \textit{vent}(\textit{Plant}) = \text{CLOSED})) \\
&\quad \Rightarrow \textit{post-Service}
\end{aligned}
$$

From this and the above results it can be deduced that:-

$\{\textit{pre-Service}\}\ (\textit{Close\_vent};\ \textit{Open\_inlet};\ \textit{Open\_outlet})\ \{\textit{post-Service}\}$

C.4. An Example of Decomposition into Conditionals.

In order to illustrate decomposition into conditionals consider an operation for which consults a sensor and then sets a flag according to the measurement. The state variables are:-

state *Detector* of

pressure : *sensor*

serve_flag : *flag*

end

sensor : **R**

flag = { SERVE, SHUTDOWN }

The operation can be defined as:-

Read_sensor ()

ext rd *pressure(Detector)* : *sensor*

wr *serve_flag(Detector)* : *flag*

pre true

post (*pressure(Detector)* $\leq$ 100 $\wedge$

serve_flag(Detector) = SERVE) $\vee$

(*pressure(Detector)* > 100 $\wedge$

serve_flag(Detector) = SHUTDOWN)

If this operation is to be decomposed into a conditional statement as follows:-

Read_sensor: if *pressure(Detector)* $\leq$ 100 then *Flag_serve*

else *Flag_shutdown*

where:

Flag_serve ()

ext rd *pressure(Detector)* : *sensor*

wr *serve_flag(Detector)* : *flag*

pre *pressure(Detector)* $\leq$ 100

post *serve_flag(Detector)* = SERVE

Flag_shutdown ()

ext rd *pressure(Detector)* : *sensor*

wr *serve_flag(Detector)* : *flag*

pre *pressure(Detector)* > 100

post *serve_flag(Detector)* = SHUTDOWN

then, as previously, there is an proof obligation to be satisfied and this takes the form:-

$$\frac{\{pre \wedge test\}TH\{post\};\{pre \wedge \neg test\}EL\{post\};pre \Rightarrow \delta_I(test)}{\{pre\}(\text{if } test \text{ then } TH \text{ else } EL)\{post\}}$$

The third requirement $\delta_I(test)$ states that the logical expression of the test condition must be defined in the implementation language. This arises because of the interpretation given to logical expression in LPF which may differ substantially from that in a programming language.

Applying this to the above example it is not difficult to see that:-

$$\begin{aligned} &pre\text{-}Read\_sensor \wedge pressure(Detector) \leq 100 \\ &\quad \Leftrightarrow pre\text{-}Flag\_serve \\ &post\text{-}Read\_sensor \Leftrightarrow (\, pressure(Detector) \leq 100 \wedge \\ &\qquad serve\_flag(Detector) = \text{SERVE}\,) \vee \\ &\qquad (\, pressure(Detector) > 100 \wedge \\ &\qquad serve\_flag(Detector) = \text{SHUTDOWN}\,) \\ &\qquad \Rightarrow post\text{-}Flag\_serve \end{aligned}$$

and

$$\begin{aligned} &pre\text{-}Read\_sensor \wedge \neg (pressure(Detector) \leq 100) \\ &\quad \Leftrightarrow pre\text{-}Flag\_shutdown \\ &post\text{-}Read\_sensor \Leftrightarrow (\, pressure(Detector) \leq 100 \wedge \\ &\qquad serve\_flag(Detector) = \text{SERVE}\,) \vee \\ &\qquad (\, pressure(Detector) > 100 \wedge \\ &\qquad serve\_flag(Detector) = \text{SHUTDOWN}\,) \\ &\qquad \Rightarrow post\text{-}Flag\_shutdown \end{aligned}$$

also the test condition is defined in most procedural languages. Therefore:-

$$\begin{aligned} &\{pre\text{-}Read\_sensor\}(\text{if } pressure(Detector) \leq 100 \text{ then } Flag\_serve \\ &\qquad \text{else } Flag\_shutdown)\{post\text{-}Read\_sensor\} \end{aligned}$$

and the decomposition is thus consistent.

C.5. Two Examples of Decomposition into Loops.

C.5.1. The inference rule for decomposition into loops.

The rule for decomposition into loops is:-

$$\frac{\{inv \wedge test\}S\{inv \wedge sofar\};\ inv \Rightarrow \delta_1(test)}{\{inv\}(\text{while } test \text{ do } S \text{ end})\{inv \wedge \neg test \wedge (sofar \vee iden)\}}$$

where:-

(a) *test*. This is the conventional end of loop condition used in procedural language iteration loops

(b) *inv*. This is an expression, called the loop invariant, which limits the states which arise in the computation. Like data type invariants it is true at all times.

(c) *sofar*. This is an expression which hold for one or more iterations. To ensure termination it is necessary that *sofar* is transitive and well-founded over the set defined by *inv*. A transitive expression is one having the following property:-

$$\forall x,y,x \cdot f(x,y) \wedge f(y,z) \Rightarrow f(x,z)$$

(d) *iden*. This is an expression which defines that the state is unchanged.

C.5.2. A simple example of decomposition into loops.

Consider the following partial specification:-

Controller ()
ext wr *pressure* : *sensor*
wr *pressure_alarm* : *flag*
pre (*pressure* ≤ 2000 ∧
pressure_alarm = OFF) ∨
(*pressure* > 2000 ∧
pressure_alarm = ON)
post *pressure* > 2000 ∧
pressure_alarm = ON

If this is to be decomposed into the code fragment:-

WHILE NOT (pressure_alarm=ON) DO

Control_pressure;

Read_alarm

END

with Control_pressure and Read_alarm defined as:-

Control_pressure ()

ext wr *pressure* : *sensor*

pre *pressure* $\leq$ 2000

post *pressure* = $\underline{pressure}$ + 10

Read_alarm ()

ext rd *pressure* : *sensor*

wr *pressure_alarm* : *flag*

pre true

post (*pressure* $\leq$ 2000 $\wedge$

pressure_alarm = OFF) $\vee$

(*pressure* > 2000 $\wedge$

pressure_alarm = ON)

Now writing:

Loop : *Control_pressure*;*Read_alarm*

gives *Loop* as:-

Loop ()

ext rd *pressure* : *sensor*

wr *pressure_alarm* : *flag*

pre *pressure* $\leq$ 2000

post *pressure* = $\underline{pressure}$ + 10 $\wedge$

(*pressure* $\leq$ 2000 $\wedge$

pressure_alarm = OFF) $\vee$

(*pressure* > 2000 $\wedge$

pressure_alarm = ON)

The decomposition is of the form:-

While *test* do

Loop

end

The major steps in the proof of this decomposition are:-

$test \triangleq \neg(pressure\_alarm = ON)$

$inv \triangleq (\ pressure \leq 2000 \wedge$

$pressure\_alarm = OFF\ ) \vee$

$(\ pressure > 2000 \wedge$

$pressure\_alarm = ON\ )$

$sofar \triangleq pressure \geq \underline{pressure} + 10$

$inv \wedge test \Leftrightarrow (\ pressure \leq 2000 \wedge$

$pressure\_alarm = OFF\ ) \vee$

$(\ pressure > 2000 \wedge$

$pressure\_alarm = ON\ )^{\bullet}$

$\neg(pressure\_alarm = ON)$

$\Leftrightarrow pressure \leq 2000$

$\Leftrightarrow pre\text{-}Loop$

$inv \wedge sofar \Leftrightarrow (\ pressure \leq 2000 \wedge$

$pressure\_alarm = OFF\ ) \vee$

$(\ pressure > 2000 \wedge$

$pressure\_alarm = ON\ )^{\bullet}$

$pressure \geq \underline{pressure} + 10$

$\Leftrightarrow post\text{-}Loop$

test is defined in most procedural languages.

Using the inference rule above gives:-

$\{inv\}$(while *test* do *Loop* end)$\{inv \wedge \neg test \wedge (sofar \vee iden)\}$

$inv \Leftrightarrow pre\text{-}Controller$

$$
\begin{aligned}
inv \wedge \neg test \wedge (sofar \vee iden) \Leftrightarrow \quad & (\, pressure \leq 2000 \wedge \\
& \quad pressure\_alarm = \mathrm{OFF}\,) \vee \\
& (\, pressure > 2000 \wedge \\
& \quad pressure\_alarm = \mathrm{ON}\,) \wedge \\
& (\, pressure\_alarm = \mathrm{ON}\,) \wedge \\
& (\, pressure \geq \underline{pressure} + 10 \vee \\
& \quad (\, pressure = \underline{pressure} \wedge \\
& pressure\_alarm = \underline{pressure\_alarm})) \\
\Rightarrow \quad & (\, pressure > 2000 \wedge \\
& \quad pressure\_alarm = \mathrm{ON}\,) \wedge \\
& (\, pressure \geq \underline{pressure} + 10 \vee \\
& \quad (\, pressure = \underline{pressure} \wedge \\
& pressure\_alarm = \underline{pressure\_alarm})) \\
\Rightarrow & \; post\text{-}Controller
\end{aligned}
$$

Therefore:

$$
\begin{aligned}
& \{pre\text{-}Controller\} \\
& \quad (\text{while } \neg(pressure = \mathrm{ON}) \text{ do} \\
& \qquad Control\_pressure; Read\_alarm \text{ end}) \\
& \qquad \{post\text{-}Controller\}
\end{aligned}
$$

and hence the decomposition is valid.

C.5.3. A more problematic example involving loops.

In this example the overall aim is the same but the definitions of Control_pressure and Read_alarm are:-

Control_pressure ()
ext wr *pressure* : *sensor*
pre *pressure* $\leq$ 2000
post true

This is unusual in that the post-condition is always true. This simply meams that the value of *pressure* is indeterminate i.e. any value within its range. *Read_alarm* remains:

Read_alarm ()

ext rd *pressure* : *sensor*

wr *pressure_alarm* : *flag*

pre true

post (*pressure* $\leq$ 2000 $\wedge$

pressure_alarm = OFF) $\vee$

(*pressure* > 2000 $\wedge$

pressure_alarm = ON)

Now in this case it is impossible to define the expression *sofar* to give a monotonically increasing value of *pressure*. Hence the termination of the loop cannot be guaranteed. Other expressions for *inv*, and *test* remain the same and hence the loop will terminate under the same conditions as above, i.e. from the loop invariant

pressure > 2000 $\wedge$

pressure_alarm = ON will be true.

APPENDIX D.

D IMPLEMENTABILITY PROOF FOR A SIMPLE LOGIC OPERATION.

D.1. The Function and its Specification.

The function of the device to be specified is a simple logical and. The device has two inputs and one output. The inputs and output each have two possible states; high or low. The device is specified as follows:

Logic = { HIGH, LOW }

And_Gate (*input1* : *Logic*; *input2* : *Logic*) *output* : *Logic*

pre *true*

post (*input1* = HIGH and
input2 = HIGH and
output = HIGH)

or ((*input1* = LOW or
input2 = LOW) and
output = LOW)

D.2. The Implementability Proof.

The purpose of implementability proofs in general is to show that for all valid inputs to an operation there exist some valid outputs. The nature proofs are discussed in more detail in Appendix B and chapter 5.

Formally the aim of the proof is to show that:-

$$\forall\, I1, I2 \in Logic \bullet pre\text{-}And\_Gate(I1,I2) \Rightarrow \exists\, O \in Logic \bullet post\text{-}And\_Gate(O,I1,I2) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ P1$$

is valid. Where, for conciseness, *I1* = *input1*, *I2* = *input2* and *O* = *output*.

To accomplish this proof it is necessary to perform a small proof about the general properties of the type *Logic*.

from $X \in$ Logic

1 X=LOW $\vee$ X=HIGH . . . . . *Logic*-defn(h)

2 $\neg\neg(X$=LOW $\vee$ X=HIGH) . . . . . $\neg\neg$-I(1)

3 $\neg(\neg(X$=LOW) $\wedge$ $\neg(X$=HIGH)) . . . . . de-M(2)

Sub-derivation(4)

4 from $\neg(\neg(X$=LOW) $\vee$ $\neg(X$=HIGH))

4.1 $\neg(X$=LOW) $\wedge$ $\neg(X$=HIGH) . . . . . de-M(h4)

infer $\neg\neg(\neg(X$=LOW) $\vee$ $\neg(X$=HIGH)) . . . . . $\neg$-I(h4,4.1,3)

5 $\neg(X$=LOW) $\vee$ $\neg(X$=HIGH) . . . . . $\neg$-E(4)

6 $\neg\neg X$=HIGH $\vee$ X=LOW . . . . . $\neg\neg$-I,comm-$\vee$(1)

7 $\neg X$=HIGH $\Rightarrow$ X=LOW . . . . . $\Rightarrow$-defn(4)

8 $\neg X$=HIGH $\vee$ $\neg X$=LOW . . . . . comm-$\vee$(3)

9 X=HIGH $\Rightarrow$ $\neg X$=LOW . . . . . $\Rightarrow$-defn(6)

10 $\neg\neg X$=LOW $\vee$ X=HIGH . . . . . $\neg\neg$-I(1)

11 $\neg X$=LOW $\Rightarrow$ X=HIGH . . . . . $\Rightarrow$-defn(8)

12 $\neg X$=LOW $\vee$ $\neg X$=HIGH . . . . . (3)

13 X=LOW $\Rightarrow$ $\neg X$=HIGH . . . . . $\Rightarrow$-defn(11)

infer $\neg X$=HIGH $\Leftrightarrow$ X=LOW, $\neg X$=LOW $\Leftrightarrow$ X=HIGH . . . . . $\Leftrightarrow$-I(5,11,7,9)

In summary this gives the following derivations which are taken as lemmas for the subsequent derivation:-

L1 $X \in$ *Logic* $\vdash$ $\neg X$=HIGH $\Leftrightarrow$ X=LOW

L2 $X \in$ *Logic* $\vdash$ $\neg X$=LOW $\Leftrightarrow$ X=HIGH

The approach to the implementability proof is to write *P1* in sequent form.

from $\forall$ *I1*, *I2* $\in$ *Logic* • *pre-And_Gate*(*I1*,*I2*) $\vdash$

$\exists$ *O* $\in$ *Logic* • *post-And_Gate*(*O*,*I1*,*I2*)

Sub-derivation(1)

1 from *I1*,*I2* $\in$ *Logic*

Sub-derivation(1.1)

1.1 from *I1*=HIGH $\wedge$ *I2*=HIGH

1.1.1 $\neg\neg$(*I1*=HIGH $\wedge$ *I2*=HIGH) . . . . . $\neg\neg$-I(h1.1)

1.1.2 $\neg$($\neg$(*I1*=HIGH) $\vee$ $\neg$(*I2*=HIGH) . . . . . de-M(1.1.1)

1.1.3 $\neg$(*I1*=LOW $\vee$ *I2*=LOW) . . . . . L1(1.1.2)

infer (*I1*=HIGH $\wedge$ *I2*=HIGH)$\Rightarrow\neg$(*I1*=LOW $\vee$ *I2*=LOW) . . . . . $\Rightarrow$-I(h1.1,1.1.3)

Sub-derivation(1.2)

1.2 from $\neg$(*I1*=LOW $\vee$ *I2*=LOW)

1.2.1 $\neg$(*I1*=LOW) $\wedge$ $\neg$(*I2*=LOW) . . . . . $\neg\neg$-I(h1.2)

1.2.2 *I1*=HIGH $\wedge$ *I2*=HIGH . . . . . L2(1.2.1)

infer $\neg$(*I1*=LOW $\vee$ *I2*=LOW)$\Rightarrow$(*I1*=HIGH $\wedge$ *I2*=HIGH) . . . . . $\vee$-I(h1.2,1.2.2)

1.3 $\neg$(*I1*=LOW $\vee$ *I2*=LOW)$\Leftrightarrow$(*I1*=HIGH $\wedge$ *I2*=HIGH) . . . . . $\Leftrightarrow$-I(1.1,1.2)

1.4 $\neg$(*I1*=LOW $\vee$ *I2*=LOW) $\vee$ (*I1*=LOW $\vee$ *I2*=LOW) . . . . . excluded middle

1.5 (*I1*=HIGH $\wedge$ *I2*=HIGH) $\vee$ (*I1*=LOW $\vee$ *I2*=LOW) . . . . . $\equiv$-subst(1.3,1.4)

Sub-derivation(1.6)

1.6 from *I1*=HIGH $\wedge$ *I2*=HIGH

Sub-derivation(1.6.1)

1.6.1 from *O*=HIGH

1.6.1.1 (*I1*=HIGH $\wedge$ *I2*=HIGH) $\wedge$ *O*=HIGH . . . . . $\wedge$-I(h1.6,h1.6.1)

1.6.1.2 ((*I1*=HIGH $\wedge$ *I2*=HIGH) $\wedge$ *O*=HIGH) $\vee$
((*I1*=LOW $\vee$ *I2*=LOW) $\wedge$ *O*=LOW) . . . . . $\vee$-I(1.6.1.1)

infer *post-And_Gate*(*O*,*I1*,*I2*) . . . . . *And_Gate*-defn(1.6.1.2)

1.6.2 HIGH $\in$ *Logic* . . . . . *Logic*-defn

infer $\exists$ *O* $\in$ *Logic* • *post-And_Gate*(*O*,*I1*,*I2*) . . . . . $\exists$-I(1.6.1,1.6.2)

Sub-derivation(1.7)

1.7 from $I1 = \text{LOW} \vee I2 = \text{LOW}$

Subderivation(1.7.1)

1.7.1 from $O = \text{LOW}$

1.7.1.1 $(I1 = \text{LOW} \vee I2 = \text{LOW}) \wedge O = \text{LOW}$ . . . . . $\wedge$-I(h1.7,h1.7.1)

1.7.1.2 $((I1 = \text{HIGH} \wedge I2 = \text{HIGH}) \wedge O = \text{HIGH}) \vee$
$((I1 = \text{LOW} \vee I2 = \text{LOW}) \wedge O = \text{LOW})$ . . . . . $\vee$-I(1.7.1.1)

infer *post-And_Gate*(O,I1,I2) . . . . . *And_Gate*-defn(1.7.1.2)

1.7.2 LOW $\in$ *Logic* . . . . . *Logic*-defn

infer $\exists\, O \in \mathit{Logic} \cdot \mathit{post\text{-}And\_Gate}(O,I1,I2)$ . . . . . $\exists$-I(1.7.1,1.7.2)

1.8 $\exists\, O \in \mathit{Logic} \cdot \mathit{post\text{-}And\_Gate}(O,I1,I2)$ . . . . . $\vee$-E(1.5,1.6,1.7)

infer $\mathit{pre\text{-}And\_Gate}(I1,I2) \Rightarrow$
$O \in \mathit{Logic} \cdot \mathit{post\text{-}And\_Gate}(O,I1,I2)$ . . . . . vac$\Rightarrow$-I(1.8)

infer $\forall\, I1,I2 \in \mathit{Logic} \cdot \mathit{pre\text{-}And\_Gate}(I1,I2) \Rightarrow$
$O \in \mathit{Logic} \cdot \mathit{post\text{-}And\_Gate}(O,I1,I2)$ . . . . . $\forall$-I(1)

APPENDIX E.

E. EBNF DESCRIPTION OF THE PROJECT'S FORMAL NOTATION.

E.1. Names and Literals.

UpperCaseCharacter = "A"|"B"|.....|"Y"|"Z".
LowerCaseCharacter = "a"|"b"|.....|"y"|"z".
NonZeroDigit = "1"|"2"|.....|"8"|"9".
Digit = "0"|NonZeroDigit.
Integer = NonZeroDigit {Digit}.
Character = UpperCaseCharacter | LowerCaseCharacter.
Ident = Character { Character | Digit |"_"|".".
QualIdent = Ident "(" Ident ")".
GeneralIdent = Ident | QualIdent.
InitialIdent = QualIdent "'".
GeneralPostIdent = GeneralIdent | GeneralIdent "'".
GrammarGraphName = "vdm" Integer.

E.2. Types.

TypeDefinition = Ident Type.
Type = "=" Enumeration | ":" Ident.
Enumeration = "{" IdentList "}".
IdentList = Ident { "," Ident }.

E.3. Expressions.

Expression = Relation | Relation "or" Expression | Relation "and"
Expression.

Relation = Equivalence | BracketedExpression.
Equivalence = GeneralIdent "=" GeneralIdent.
BracketedExpression = "(" Expression ")".

PostExpression = PostRelation | PostRelation "or" PostExpression |
PostRelation "and" PostExpression.
PostRelation = PostEquivalence : BracketedPostExpression.
PostEquivalence = GeneralPostIdent "=" GeneralPostIdent.
BracketedPostExpression = "(" PostExpression ")".

VariableDefinition = Ident ":" Ident.

E.4. Statements.

StatementBlock = Sequence | Selection | Iteration.

Sequence = "SEQ" SequenceStructure "END".
SequenceStructure = OperationText { StatementBlock }.
OperationText = "DO" Integer ":" OperationDefinition

Selection = "SEL" SelectionStructure { AlternativeStructure } "END".
SelectionStructure = "(C" Integer "):" Expression "end"
StatementBlock.
AlternativeStructure = "ALT" SelectionStructure.

Iteration = "ITR" IterationStructure "END".
IterationStructure = IterationHeader StatementBlock.
IterationHeader = IterationType "(C" Integer "):" Expression "end".
IterationType = "W" | "F" | "U".

E.5. Definitions.

OperationDefinition = Signature Opbody "endop".

Signature = Ident Arguments.

Arguments = "(" [InputArgs] ")" [OutputArgs].

InputArgs = VarDefine { ";" VarDefine }.

OutputArgs = VarDefine { ";" VarDefine }.

OpBody = ExternalList PreCondition PostCondition.

ExternalList = "ext" { ReadExt | WriteExt }.

ReadExt = "rd" QualIdent ":" Ident.

WriteExt = "wr" QualIdent ":" Ident.

PreCondition = "pre" PreDeclare.

PreDeclare = "true" | Expression.

PostCondition = "post" PostExpression "endpost".

VarDefine = Ident ":" Ident.

SimpleVar = TypeDeclaration.

SimpleVarList = SimpleVar { SimpleVar }.

CompositeVar = "compose" Ident "of" SimpleVarList "end".

VarDefineList = CompositeVar { SimpleVar }.

Definitions = GrammarGraphName VarDefineList "enddefine".

StateDefinition = "state" [VarDefine].

InitialStateDefinition = "initial" Expression "end".

SpecBody = StateDefinition InitialStateDefinition StatementBlock.

Specification = Definitions SpecBody

APPENDIX F.

F. THE GRAMMAR TREE FOR THE PARSING OF THE FORMAL NOTATION.

/* The number of nodes in the tree

248

/* The grammar tree

```
  100  action  101     0  specification
  101  n       102   104  !
  102  action  103     0  definitions
  103  n      2000     0  !
  104  n       105     0  !
  105  action  106     0  specbody
  106  n      1000  4000  !

 1000  action 1001     0  initial_state
 1001  t      1001  1002  state
 1002  n      1100  1300  !

 1100  action 1101     0  state
 1101  a      1102  1112  !
 1102  n      1103  1106  !
 1103  action 1104     0  varname
 1104  action 1105     0  identifier
 1105  t      1105     0  identifier
 1106  t      1106  1107  :
 1107  n      1108  1111  !
 1108  action 1109     0  typename
 1109  action 1110     0  identifier
 1110  t      1110     0  identifier
 1111  t      1111     0  initial
 1112  t      1112     0  initial
```

```
1300  action 1301     0  initial_clause
1301  action 1302     0  analyse
1302  t      1302     0  end

2000  action 2001     0  variablelist
2001  a      2002  2006  !
2002  n      2100  2003  !
2003  a      2004  2005  !
2004  t      2004     0  enddefine
2005  n      2000     0  !
2006  t      2006  2007  compose
2007  n      2200  2008  !
2008  a      2009  2010  !
2009  t      2009     0  enddefine
2010  n      2000     0  !

2100  action 2101     0  simplevar
2101  n      2102  2105  !
2102  action 2103     0  varname
2103  action 2104     0  identifier
2104  t      2104     0  identifier
2105  a      2106  2111  !
2106  t      2106  2107  :
2107  n      2108     0  !
2108  action 2109     0  typename
2109  action 2110     0  identifier
2110  t      2110     0  identifier
2111  t      2111  2112  =
2112  t      2112  2113  {
2113  n      2150     0  !

2150  action 2151     0  elementlist
2151  n      2152  2155  !
2152  action 2153     0  elementname
2153  action 2154     0  identifier
2154  t      2154     0  identifier
2155  a      2156  2158  !
```

```
2156  t       2156   2157  ,
2157  n       2150      0  !
2158  t       2158      0  }

2200  action  2201      0  compvar
2201  n       2202   2205  !
2202  action  2203      0  varname
2203  action  2204      0  identifier
2204  t       2204      0  identifier
2205  t       2205   2206  of
2206  n       2207      0  !
2207  action  2208      0  simplevarlist
2208  n       2100   2209  !
2209  a       2210   2211  !
2210  t       2210      0  end
2211  n       2207      0  !

3000  action  3001      0  operation
3001  n       3100   3002  !
3002  n       3003   3005  !
3003  action  3004      0  body
3004  n       3200   3300  !
3005  t       3005      0  endop

3100  action  3101      0  signature
3101  n       3102   3105  !
3102  action  3103      0  opname
3103  action  3104      0  identifier
3104  t       3104      0  identifier
3105  n       3106      0  !
3106  action  3107      0  arguments
3107  t       3107   3108  (
3108  n       3120   3160  !

3120  action  3121      0  input
3121  a       3122   3123  !
3122  t       3122      0  )
```

```
3123  n       3124   3134  !
3124  action  3125      0  type_declare
3125  n       3126   3129  !
3126  action  3127      0  varname
3127  action  3128      0  identifier
3128  t       3128      0  identifier
3129  t       3129   3130  :
3130  n       3131      0  !
3131  action  3132      0  typename
3132  action  3133      0  identifier
3133  t       3133      0  identifier
3134  a       3135   3136  !
3135  t       3135      0  )
3136  t       3136   3137  ;
3137  n       3120      0  !

3160  action  3161      0  output
3161  a       3162   3163  !
3162  t       3162      0  ext
3163  n       3164   3174  !
3164  action  3165      0  type_declare
3165  n       3166   3169  !
3166  action  3167      0  varname
3167  action  3168      0  identifier
3168  t       3168      0  identifier
3169  t       3169   3170  :
3170  n       3171      0  !
3171  action  3172      0  typename
3172  action  3173      0  identifier
3173  t       3173      0  identifier
3174  a       3175   3176  !
3175  t       3175      0  ext
3176  t       3176   3177  ;
3177  n       3160      0  !

3200  action  3201      0  ext_list
3201  a       3202   3203  !
```

```
3202  t       3202     0  pre
3203  n       3204  3208  !
3204  action  3205     0  declare
3205  a       3206  3207  !
3206  n       3230     0  !
3207  n       3260     0  !
3208  a       3209  3210  !
3209  t       3209     0  pre
3210  n       3200     0  !

3230  action  3231     0  read
3231  t       3231  3232  rd
3232  n       3233     0  !
3233  action  3234     0  type_declare
3234  n       3235  3238  !
3235  action  3236     0  varname
3236  action  3237     0  identifier
3237  t       3237     0  identifier
3238  t       3238  3239  :
3239  n       3240     0  !
3240  action  3241     0  typename
3241  action  3242     0  identifier
3242  t       3242     0  identifier

3260  action  3261     0  write
3261  t       3261  3262  wr
3262  n       3263     0  !
3263  action  3264     0  type_declare
3264  n       3265  3268  !
3265  action  3266     0  varname
3266  action  3267     0  identifier
3267  t       3267     0  identifier
3268  t       3268  3269  :
3269  n       3270     0  !
3270  action  3271     0  typename
3271  action  3272     0  identifier
3272  t       3272     0  identifier
```

```
3300  action 3301     0  conditions
3301  n      3302  3309  !
3302  action 3303     0  pre_clause
3303  a      3304  3307  !
3304  action 3305     0  true
3305  t      3305  3306  true
3306  t      3306     0  post
3307  action 3308     0  analyse
3308  t      3308     0  post
3309  n      3310     0  !
3310  action 3311     0  post_clause
3311  action 3312     0  analyse
3312  t      3312     0  endpost

4000  action 4001     0  block
4001  a      4002  4003  !
4002  t      4002  4100  SEQ
4003  a      4004  4005  !
4004  t      4004  4200  SEL
4005  t      4005  4400  ITR

4100  action 4101     0  sequence
4101  n      4102     0  !
4102  a      4103  4110  !
4103  n      4104  4107  !
4104  t      4104  4105  DO
4105  t      4105  4106  identifier
4106  t      4106  3000  :
4107  n      4150     0  !

4110  a      4111  4112  !
4111  n      4000  4150  !
4112  t      4112     0  END

4150  a      4151  4152  !
4151  t      4151     0  END
4152  action 4153     0  continuation
```

```
4153  n       4000   4150  !

4200  action  4201      0  selection
4201  n       4202   4207  !
4202  action  4203      0  condition
4203  t       4203   4204  (
4204  t       4204   4205  identifier
4205  action  4206      0  analyse
4206  t       4206      0  end
4207  action  4208      0  selection_body
4208  a       4209   4213  !
4209  n       4000   4210  !
4210  a       4211   4212  !
4211  t       4211      0  END
4212  t       4212   4300  ALT
4213  a       4214   4215  !
4214  t       4214      0  END
4215  t       4215   4300  ALT

4300  action  4301      0  alternative
4301  n       4302   4307  !
4302  action  4303      0  condition
4303  t       4303   4304  (
4304  t       4304   4305  identifier
4305  action  4306      0  analyse
4306  t       4306      0  end
4307  action  4308      0  alternative_body
4308  a       4309   4313  !
4309  n       4000   4310  !
4310  a       4311   4312  !
4311  t       4311      0  END
4312  t       4312   4300  ALT
4313  a       4314   4315  !
4314  t       4314      0  END
4315  t       4315   4300  ALT

4400  action  4401      0  iteration
```

```
4401  n       4402   4417  !
4402  action  4403      0  iteration_header
4403  n       4404   4412  !
4404  a       4405   4407  !
4405  action  4406      0  while
4406  t       4406      0  W
4407  a       4408   4410  !
4408  action  4409      0  until
4409  t       4409      0  U
4410  action  4411      0  for
4411  t       4411      0  F
4412  t       4412   4413  (
4413  t       4413   4414  identifier
4414  action  4415      0  condition
4415  action  4416      0  analyse
4416  t       4416      0  end
4417  action  4418      0  iteration_body
4418  a       4419   4421  !
4419  n       4000   4420  !
4420  t       4420      0  END
4421  t       4421      0  END

/* Key words.

SEQ
SEL
ALT
ITR
END
DO
U
F
W
:
endspec
enddefine
compose
```

```
of
end
=
{
,
:
}
state
initial
endop
)
(
;
ext
pre
wr
rd
true
post
endpost
and
or
!
%
```

APPENDIX G.

G. ANIMATION PROTOTYPE OF A LOGIC GATE.

G.1. Statement of Requirements for the Logic Gate.

The logic gate is specified as follows:-

* The logic gate has two input lines and one output line.
* These lines may be either HIGH or LOW.
* If both input lines are HIGH then the output shall be HIGH.
* In other cases the output line should be LOW.

G.2. Formal Specification of the Logic Gate.

The formal specification of this gate is:-

```
/* Current version of VDM subset
vdm5

/* Data type definition

Logic = { High , Low }

enddefine

/* State type declaration

state

/* Initial value details

initial
```

```
end

/* Operation specification
And_Gate ( Input1 : Logic ; Input2 : Logic ) Output : Logic

/* No External Effects
ext

/* Operation defined for all input values
pre true

/* Relationship between input and output variables
post     (     Input1 = High
          and Input2 = High
          and Output = High )
      or (     (    Input1 = Low
                or Input2 = Low )
          and Output = Low )
endpost

endop

endspec
!
```

G.3. Animation Code Produced by the Animation Process.

The resulting output from the animator is :-

```
PREAMBLE

   DEFINE ..High TO MEAN 1
   DEFINE ..Low TO MEAN 2

END
```

```
ROUTINE And_Gate GIVEN Input1,
                       Input2
                YIELDING Output

   DEFINE Input1,
          Input2,
          Output
          AS INTEGER VARIABLES

   CALL Tim.Update.Display GIVEN And_Gate,
                                 "Pre "

   CALL Tim.Local.Update.And_Gate GIVEN Input1,
                                  Input2,
                                  0

      IF (     Input1 = ..High
          AND Input2 = ..High )
          LET Output = ..High
      ENDIF

      IF (    Input1 = ..Low
          OR Input2 = ..Low )
          LET Output = ..Low
      ENDIF

   CALL Tim.Update.Display GIVEN And_Gate,
                                 "Post"

   CALL Tim.Local.Update.And_Gate GIVEN Input1,
                                  Input2,
                                  Output

END
```

```
ROUTINE Tim.Local.Update.And_Gate GIVEN Input1,
                                  Input2,
```

```
                                Output
DEFINE Input1,
       Input2,
       Output AS INTEGER VARIABLES

SELECT CASE Input1

   CASE ..High
      ** Update appropriate icon **
   CASE ..Low
      ** Update appropriate icon **
   CASE 0

ENDSELECT


SELECT CASE Input2

   CASE ..High
      ** Update appropriate icon **
   CASE ..Low
      ** Update appropriate icon **
   CASE 0

ENDSELECT


SELECT CASE Output

   CASE ..High
      ** Update appropriate icon **
   CASE ..Low
      ** Update appropriate icon **
   CASE 0

ENDSELECT
```

```
END

MAIN


END


ROUTINE Tim.Update.Display GIVEN Op.Name, Condition

   DEFINE Op.Name, Condition AS TEXT VARIABLES
   LET DTVAL.A(DFIELD.F( ** Op Box Name ** , ** Main.Form
** )
            = Op.Name
   DISPLAY DFIELD.F( ** Op Box Name ** , ** Main.Form ** )
   LET  DTVAL.A(DFIELD.F(**  Condition  Box  Name  **,**
Main.Form **)
            = Condition
   DISPLAY  DFIELD.F(  **  Condition  Box  Name  **  ,  **
Main.Form **)

END
```

APPENDIX H.

H. THE NITROGEN/HYDROGEN COMPRESSOR PLANT.

H.1. The Statement of Requirements.

The system described here is a large chemical plant. Its purpose is to compress nitrogen. The compressed nitrogen is then stored in liquid form in a storage vessel. Nitrogen is drawn from the storage vessel for use in other processes.

The system consists of two compressors (J1 and J2), a storage vessel, a number of valves and a number of alarms. The plant schematic diagram in Figure 26 shows the layout of these components.

The input signals available to the control system are:-

Pressure Sensors	Possible States
Extra Low	Silent/Ringing
Low	Silent/Ringing
High	Silent/Ringing
Extra High	Silent/Ringing

Compressor Alarms

J1 Alarm	Silent/Ringing
J2 Alarm	Silent/Ringing

The output lines used by the control system are:-

Compressor Signals

- J1 Start
- J2 Start
- J1 Stop
- J2 Stop

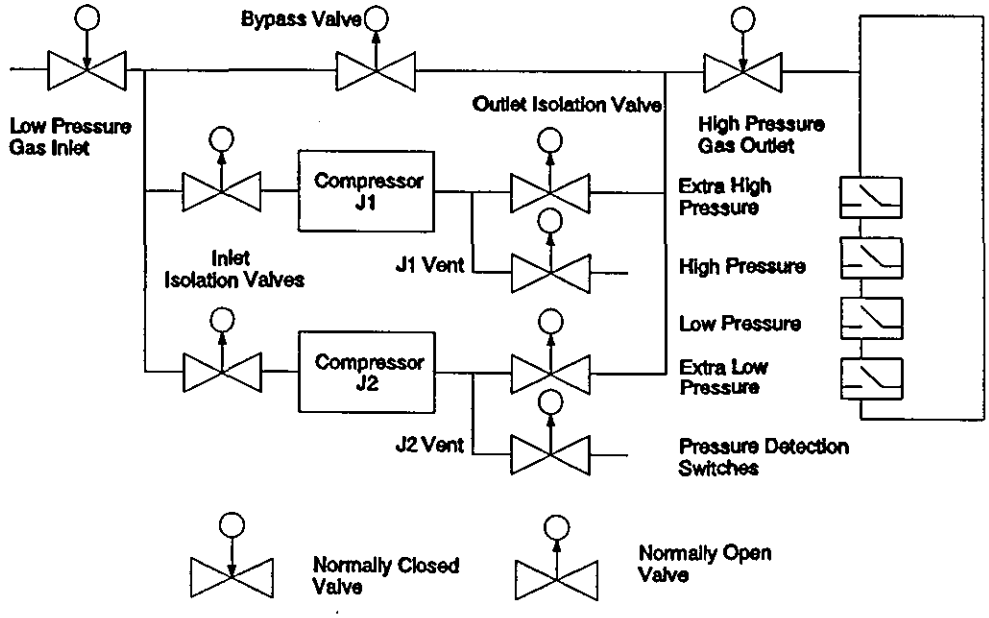


Figure 26 Plant Schematic.

Valve Signals

Open Inlet Valve
Open Outlet Valve
Open Bypass Valve
Open J1 Isolation Valves
Open J2 Isolation Valves
Open J1 Vent Valve
Open J2 Vent Valve

Close Inlet Valve
Close Outlet Valve
Close Bypass Valve
Close J1 Isolation Valves
Close J2 Isolation Valves
Close J1 Vent Valve
Close J2 Vent Valve

The controller uses a number of overrides:-

Overrides	Possible States
Extra Low Pressure O/R	Set/Reset
Low Pressure O/R	Set/Reset
J1 Low Pressure Oil O/R	Set/Reset
J2 Low Pressure Oil O/R	Set/Reset

Note the compressor alarms have two parts. The first is the Low Pressure Oil alarm, this may be overridden. The second part includes a number of alarms such as high pressure and temperature alarms, this may not be overridden. These individual alarms may all cause the compressor alarm to ring.

The system operates cyclically. One compressor is used to raise the pressure in the storage vessel from the low pressure level to the high pressure level. The compressor is then switched off. As gas is drawn from the storage vessel the pressure falls. When it reaches the low pressure level the other compressor is used to repeat the procedure.

1. At power on or in the event of a power failure the system reverts to its fail safe condition. That is:-

   a) Both compressors are off.
   b) The Inlet and Outlet valves are closed.
   c) The Bypass valve is open.
   d) The isolation and vent valves on both compressors are open.

2. The plant must be shut down if:-

   a) The Extra High Pressure alarm is ringing.
   b) The Manual Stop button is pressed.
   c) Both compressors fail.

3. A compressor has failed if either of the following is true:-

   a) Its alarm is ringing.
   b) The compressor is running and the Extra Low Pressure alarm is ringing.

4. A compressor which has broken down may not be used again until it has been repaired.

5. Only one compressor should be running at a time.

6. When a compressor is running, the compressor which is not being used must be isolated. A compressor is isolated if all the following conditions are met :-

   a) It is off.
   b) Its isolation valves are closed.
   c) Its vent valve is open.
   d) Its Low Pressure Oil alarm is overridden.

7. The start up procedure for a compressor is:-

   a) Open the compressor isolation valves.
   b) Wait for Low Pressure alarm to ring.
   c) Close the compressor vent valve.
   d) Start the compressor.
   e) After 30 seconds, remove the Low Pressure Oil alarm override.
   f) After 40 seconds, close the Bypass valve.

g) Open the Inlet and Outlet valves.

This process can be aborted at any stage if the compressor alarm begins to ring. To shut the compressor down safely the appropriate part of section 8. should be used.

The initial start up procedure is slightly different, as it has to take account of there being very little pressure in the storage vessel. Two operations are added. Firstly the Extra Low Pressure alarm is overridden. The procedure then continues as normal until the end when there is a further wait of 3 minutes, when the Extra Low Pressure alarm override is removed.

8. The complete shut down procedure for a compressor is:-

a) The compressor Low Pressure Oil alarm is overridden.
b) The compressor is switched off.
c) The Inlet and Outlet valves are closed.
d) The compressor vent valve is opened.
e) After 40 seconds, the Bypass valve is opened.
f) After a further 20 seconds, the compressor Isolation valves are closed.

A partial shut down of a compressor must use parts a and b, the other parts will depend on the point at which the compressor breaks down.

9. The end of a compression cycle is indicated by:-

a) The plant needing to be shut down (see 2 above).
b) The compressor breaking down (see 3 above).
c) The High Pressure alarm ringing.

H.2. A Formal Specification of The Plant Operation.

H.2.1. State and type specification.

An informal description of the system is given above. The formal specification arrived at is given below.

Using the subset gives the following data type definitions:-

```
/* The current version of the specification language:

vdm5

/* The type for defining the state

compose PlantState of
     Bypass : Valve
     Isol1 : Valve
     Isol2 : Valve
     Vent1 : Valve
     Vent2 : Valve
     InOutlet : Valve
     EHP : Alarm
     HP : Alarm
     LP : ORAlarm
     ELP : ORAlarm
     Compressor : SelectedCompressor
     Comp1 : CompState
     Comp2 : CompState
     Comp1Lock : Lockout
     Comp2Lock : Lockout
     Comp1Alarm : ORAlarm
     Comp2Alarm : ORAlarm
     OperatorSwitch : Switch
end

/*  Associated data types

     Valve = { OPEN , CLOSED }
     Alarm = { ON , OFF , UNKNOWN }
     ORAlarm = { ON , OFF , OVERRIDE , UNKNOWN }
     SelectedCompressor = { COMP1 , COMP2 , NONE }
     CompState = { ON , OFF }
     Lockout = { SET , RESET }
```

```
     Switch = { ON , OFF , UNKNOWN }
enddefine

The initial state is also a fail safe state.  The state and
its initial value are as follows:-

/* State variable name and type

state
     PState : PlantState

/* Initial state

initial
       ( Bypass(PState) = OPEN
       and Isol1(PState) = OPEN
       and Isol2(PState) = OPEN
       and Vent1(PState) = OPEN
       and Vent2(PState) = OPEN
       and InOutlet(PState) = CLOSED
       and EHP(PState) = UNKNOWN
       and HP(PState) = UNKNOWN
       and LP(PState) = UNKNOWN
       and ELP(PState) = UNKNOWN
       and Compressor(PState) = NONE
       and Comp1(PState) = OFF
       and Comp2(PState) = OFF
       and Comp1Lock(PState) = RESET
       and Comp2Lock(PState) = RESET
       and Comp1Alarm(PState) = UNKNOWN
       and Comp2Alarm(PState) = UNKNOWN
       and OperatorSwitch(PState) = UNKNOWN )
end
```

H.2.2. Specification structure.

The structure diagram for the specification is shown in the figure (Figure 27) below.

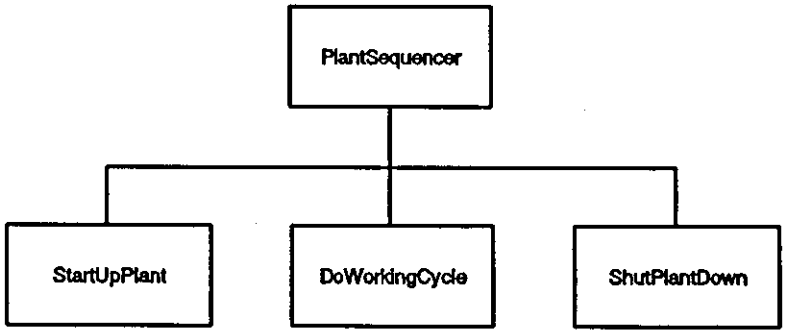


Figure 27 Structure of the Plant Controller Specification.

H.2.3. Specification of the operation PlantSequencer.

At the highest level the system is defined as only a single operation, specifying the desired start and end conditions. In this way the requirements for these are explicitly established i.e. whatever happens during the plant's working, the control system must ensure that the plant ends in the fail safe condition. The operation is specified as follows:-

```
SEQ
DO 1 :

PlantSequencer ( )

ext
     wr Bypass(PState) : Valve
     wr Isol1(PState) : Valve
     wr Isol2(PState) : Valve
     wr Vent1(PState) : Valve
     wr Vent2(PState) : Valve
     wr InOutlet(PState) : Valve
     wr EHP(PState) : Alarm
     wr HP(PState) : Alarm
     wr LP(PState) : ORAlarm
     wr ELP(PState) : ORAlarm
     wr Compressor(PState) : SelectedCompressor
     wr Comp1(PState) : CompState
     wr Comp2(PState) : CompState
     wr Comp1Lock(PState) : Lockout
     wr Comp2Lock(PState) : Lockout
     wr Comp1Alarm(PState) : ORAlarm
     wr Comp2Alarm(PState) : ORAlarm
     wr OperatorSwitch(PState) : Switch

pre
     Bypass(PState) = OPEN and
     Isol1(PState) = OPEN and
     Isol2(PState) = OPEN and
```

```
      Vent1(PState) = OPEN and
      Vent2(PState) = OPEN and
      InOutlet(PState) = CLOSED and
      EHP(PState) = UNKNOWN and
      HP(PState) = UNKNOWN and
      LP(PState) = UNKNOWN and
      ELP(PState) = UNKNOWN and
      Compressor(PState) = NONE and
      Comp1(PState) = OFF and
      Comp2(PState) = OFF and
      Comp1Lock(PState) = RESET and
      Comp2Lock(PState) = RESET and
      Comp1Alarm(PState) = UNKNOWN and
      Comp2Alarm(PState) = UNKNOWN and
      OperatorSwitch(PState) = UNKNOWN

post
      Bypass(PState) = OPEN and
      Isol1(PState) = OPEN and
      Vent1(Pstate) = OPEN and
      Isol2(PState) = OPEN and
      Vent2(PState) = OPEN and
      InOutlet(PState) = CLOSED and
      Comp1(PState) = OFF and
      Comp2(PState) = OFF and
      EHP(PState) = UNKNOWN and
      HP(PState) = UNKNOWN and
      LP(PState) = UNKNOWN and
      ELP(PState) = UNKNOWN and
      Compressor(PState) = NONE and
      Comp1(PState) = OFF and
      Comp2(PState) = OFF and
      Comp1Lock(PState) = RESET and
      Comp2Lock(PState) = RESET and
      Comp1Alarm(PState) = UNKNOWN and
      Comp2Alarm(PState) = UNKNOWN and
      OperatorSwitch(PState) = UNKNOWN
```

```
endpost
endop
```

This single operation is then decomposed into a sequence of three further operation:-

StartUpPlant
DoWorkingCycle
ShutplantDown

H.2.4. Specification of the operation StartUpPlant.

The operation StartUpPlant describes the start up of the plant when there is no high pressure gas in the system. There are three possible outcomes of this operation. These are represented by the three disjuncts in the post condition. They three cases are:-

(a) The first compressor (Compressor1) starts successfully and begins to compress gas correctly. The other compressor (Compressor2) remains switched off and isolated from the high pressure gas.

(b) The first compressor fails to start or compress gas correctly and is switched off, isolated and has its Lock flag set. The second compressor then starts successfully and begins to compress gas correctly.

(c) Both compressors fail in some way and are switched off, isolated and have their Lock flags set.

The specification of this operation is as follows:-

```
SEQ
DO 2 :

StartUpPlant ( )

ext
     wr Bypass(PState) : Valve
     wr Isol1(PState) : Valve
     wr Isol2(PState) : Valve
```

```
    wr Vent1(PState) : Valve
    wr Vent2(PState) : Valve
    wr InOutlet(PState) : Valve
    wr EHP(PState) : Alarm
    wr HP(PState) : Alarm
    wr LP(PState) : ORAlarm
    wr ELP(PState) : ORAlarm
    wr Compressor(PState) : SelectedCompressor
    wr Comp1(PState) : CompState
    wr Comp2(PState) : CompState
    wr Comp1Lock(PState) : Lockout
    wr Comp2Lock(PState) : Lockout
    wr Comp1Alarm(PState) : ORAlarm
    wr Comp2Alarm(PState) : ORAlarm
    wr OperatorSwitch(PState) : Switch

pre
    Bypass(PState) = OPEN and
    Isol1(PState) = OPEN and
    Isol2(PState) = OPEN and
    Vent1(PState) = OPEN and
    Vent2(PState) = OPEN and
    InOutlet(PState) = CLOSED and
    EHP(PState) = UNKNOWN and
    HP(PState) = UNKNOWN and
    LP(PState) = UNKNOWN and
    ELP(PState) = UNKNOWN and
    Compressor(PState) = NONE and
    Comp1(PState) = OFF and
    Comp2(PState) = OFF and
    Comp1Lock(PState) = RESET and
    Comp2Lock(PState) = RESET and
    Comp1Alarm(PState) = UNKNOWN and
    Comp2Alarm(PState) = UNKNOWN and
    OperatorSwitch(PState) = UNKNOWN

post
```

```
     ( Compressor(PState) = COMP1 and
     Comp1Lock(PState) = RESET and
     Comp1(PState) = ON and
     Isol1(PState) = OPEN and
     Vent1(PState) = CLOSED and
     Bypass(PState) = CLOSED and
     InOutlet(PState) = OPEN and
     EHP(PState) = OFF and
     HP(PState) = OFF and
     LP(PState) = OFF and
     ELP(PState) = OFF and
     Comp1Alarm(PState) = OFF and
     Comp2Lock(PState) = RESET and
     Comp2(PState) = OFF and
     Isol2(PState) = CLOSED and
     Vent2(PState) = OPEN and
     Comp2Alarm(PState) = OVERRIDE and
     OperatorSwitch(PState) = ON )

or   ( Compressor(PState) = COMP2 and
     Comp2Lock(PState) = RESET and
     Comp2(PState) = ON and
     Isol2(PState) = OPEN and
     Vent2(PState) = CLOSED and
     Bypass(PState) = CLOSED and
     InOutlet(PState) = OPEN and
     EHP(PState) = OFF and
     HP(PState) = OFF and
     LP(PState) = OFF and
     ELP(PState) = OFF and
     Comp2Alarm(PState) = OFF and
     Comp1Lock(PState) = SET and
     Comp1(PState) = OFF and
     Isol1(PState) = CLOSED and
     Vent1(PState) = OPEN and
     Comp1Alarm(PState) = OVERRIDE and
     OperatorSwitch(PState) = ON )
```

```
or   ( Compressor(PState) = NONE and
     Comp1Lock(PState) = SET and
     Comp2Lock(PState) = SET and
     Comp1(PState) = OFF and
     Comp2(PState) = OFF and
     Isol1(PState) = OPEN and
     Vent1(PState) = OPEN and
     Isol2(PState) = OPEN and
     Vent2(PState) = OPEN and
     Bypass(PState) = OPEN and
     InOutlet(PState) = CLOSED and
     EHP(PState) = OFF and
     HP(PState) = OFF and
     LP(PState) = OFF and
     ELP(PState) = OFF and
     Comp1Alarm(PState) = OVERRIDE and
     Comp2Alarm(PState) = OVERRIDE and
     OperatorSwitch(PState) = ON )

endpost
endop
END
```

H.2.5. Specification of the operation DoWorkingCycle.

The operation DoWorkingCycle describes the normal working cycle of the plant during which gas is compressed using alternate compressors. There are four possible outcomes of this operation:-

(a) At some time during the cycle both compressors fail. They are switched off and isolated.

(b) The operator switch is moved to the OFF position. Both compressors must be switched off and isolated.

(c) The extra high pressure alarm begins to ring. This is an exceedingly dangerous condition. Both compressors must be switched off and isolated.

(d) Both compressors failed during StartUp. The working cycle therefore does nothing.

The specification of this operation is as follows:-

```
SEQ
DO 3 :

DoWorkingCycle ( )

ext
      wr Bypass(PState) : Valve
      wr Isol1(PState) : Valve
      wr Isol2(PState) : Valve
      wr Vent1(PState) : Valve
      wr Vent2(PState) : Valve
      wr InOutlet(PState) : Valve
      wr EHP(PState) : Alarm
      wr HP(PState) : Alarm
      wr LP(PState) : ORAlarm
      wr ELP(PState) : ORAlarm
      wr Compressor(PState) : SelectedCompressor
      wr Comp1(PState) : CompState
      wr Comp2(PState) : CompState
      wr Comp1Lock(PState) : Lockout
      wr Comp2Lock(PState) : Lockout
      wr Comp1Alarm(PState) : ORAlarm
      wr Comp2Alarm(PState) : ORAlarm
      wr OperatorSwitch(PState) : Switch

pre
      ( Compressor(PState) = COMP1 and
      Comp1Lock(PState) = RESET and
      Comp1(PState) = ON and
      Isol1(PState) = OPEN and
      Vent1(PState) = CLOSED and
      Bypass(PState) = CLOSED and
      InOutlet(PState) = OPEN and
      EHP(PState) = OFF and
      HP(PState) = OFF and
```

```
    LP(PState) = OFF and
    ELP(PState) = OFF and
    Comp1Alarm(PState) = OFF and
    Comp2Lock(PState) = RESET and
    Comp2(PState) = OFF and
    Isol2(PState) = CLOSED and
    Vent2(PState) = OPEN and
    Comp2Alarm(PState) = OVERRIDE and
    OperatorSwitch(PState) = ON )

or  ( Compressor(PState) = COMP2 and
    Comp2Lock(PState) = RESET and
    Comp2(PState) = ON and
    Isol2(PState) = OPEN and
    Vent2(PState) = CLOSED and
    Bypass(PState) = CLOSED and
    InOutlet(PState) = OPEN and
    EHP(PState) = OFF and
    HP(PState) = OFF and
    LP(PState) = OFF and
    ELP(PState) = OFF and
    Comp2Alarm(PState) = OFF and
    Comp1Lock(PState) = SET and
    Comp1(PState) = OFF and
    Isol1(PState) = CLOSED and
    Vent1(PState) = OPEN and
    Comp1Alarm(PState) = OVERRIDE and
    OperatorSwitch(PState) = ON )

or  ( Compressor(PState) = NONE and
    Comp1Lock(PState) = SET and
    Comp2Lock(PState) = SET and
    Comp1(PState) = OFF and
    Comp2(PState) = OFF and
    Isol1(PState) = OPEN and
    Vent1(PState) = OPEN and
    Isol2(PState) = OPEN and
```

```
      Vent2(PState) = OPEN and
      Bypass(PState) = OPEN and
      InOutlet(PState) = CLOSED and
      EHP(PState) = OFF and
      HP(PState) = OFF and
      LP(PState) = OFF and
      ELP(PState) = OFF and
      Comp1Alarm(PState) = OVERRIDE and
      Comp2Alarm(PState) = OVERRIDE and
      OperatorSwitch(PState) = ON )

post

      ( Compressor(PState)' = NONE and
      Bypass(PState) = Bypass(PState)' and
      Isol1(PState) = Isol1(PState)' and
      Isol2(PState) = Isol2(PState)' and
      Vent1(PState) = Vent1(PState)' and
      Vent2(PState) = Vent2(PState)' and
      InOutlet(PState) = InOutlet(PState)' and
      EHP(PState) = EHP(PState)' and
      HP(PState) = HP(PState)' and
      LP(PState) = LP(PState)' and
      ELP(PState) = ELP(PState)' and
      Compressor(PState) = Compressor(PState)' and
      Comp1(PState) = Comp1(PState)' and
      Comp2(PState) = Comp2(PState)' and
      Comp1Lock(PState) = Comp1Lock(PState)' and
      Comp2Lock(PState) = Comp2Lock(PState)' and
      Comp1Alarm(PState) = Comp1Alarm(PState)' and
      Comp2Alarm(PState) = Comp2Alarm(PState)' and
      OperatorSwitch(PState) = OperatorSwitch(PState)' )

  or  (
      (     (     Compressor(PState)' = COMP1
            or    Compressor(PState)' = COMP2 )
      and   (     (     OperatorSwitch(PState) = OFF
```

```
                and  EHP(PState) = OFF
                and  Compressor(PState)    =
Compressor(PState)')
          or    (    EHP(PState) = ON
                and  OperatorSwitch(PState)    =
                          OperatorSwitch(PState)'
                and  Compressor(PState)    =
Compressor(PState)')
          or    (    Compressor(PState) = NONE
                and  EHP(PState) = OFF
                and  OperatorSwitch(PState) =
OperatorSwitch(PState)'
                and  Comp1Lock(PState) = SET
                and  Comp2Lock(PState) = SET ) ) ) and
     HP(PState) = OFF and
     LP(PState) = OFF and
     ELP(PState) = OFF and
     Comp1(PState) = OFF and
     Comp2(PState) = OFF and
     Isol1(PState) = OPEN and
     Isol2(PState) = OPEN and
     Vent1(PState) = OPEN and
     Vent2(PState) = OPEN and
     Bypass(PState) = OPEN and
     InOutlet(PState) = CLOSED and
     Comp1Alarm(PState) = OVERRIDE and
     Comp2Alarm(PState) = OVERRIDE )

  endpost
  endop
END
```

H.2.6. Specification of the operation ShutPlantDown.

The operation ShutPlantDown ensures that at the end of the plant operation everything is returned to its fail safe position. Its specification is as follows:-

```
SEQ
DO 4 :

ShutPlantDown ( )

ext
     wr Bypass(PState) : Valve
     wr Isol1(PState) : Valve
     wr Isol2(PState) : Valve
     wr Vent1(PState) : Valve
     wr Vent2(PState) : Valve
     wr InOutlet(PState) : Valve
     wr EHP(PState) : Alarm
     wr HP(PState) : Alarm
     wr LP(PState) : ORAlarm
     wr ELP(PState) : ORAlarm
     wr Compressor(PState) : SelectedCompressor
     wr Comp1(PState) : CompState
     wr Comp2(PState) : CompState
     wr Comp1Lock(PState) : Lockout
     wr Comp2Lock(PState) : Lockout
     wr Comp1Alarm(PState) : ORAlarm
     wr Comp2Alarm(PState) : ORAlarm
     wr OperatorSwitch(PState) : Switch

pre
     ( Compressor(PState) = NONE and
     Comp1Lock(PState) = SET and
     Comp2Lock(PState) = SET and
     Comp1(PState) = OFF and
     Comp2(PState) = OFF and
     Isol1(PState) = OPEN and
     Vent1(PState) = OPEN and
     Isol2(PState) = OPEN and
     Vent2(PState) = OPEN and
     Bypass(PState) = OPEN and
     InOutlet(PState) = CLOSED and
```

```
      EHP(PState) = OFF and
      HP(PState) = OFF and
      LP(PState) = OFF and
      ELP(PState) = OFF and
      Comp1Alarm(PState) = OVERRIDE and
      Comp2Alarm(PState) = OVERRIDE and
      OperatorSwitch(PState) = ON )

or    (     (     OperatorSwitch(PState) = OFF
            and   EHP(PState) = OFF )
      or    (     EHP(PState) = ON
            and   OperatorSwitch(PState) = ON )
      and   (     Compressor(PState) = COMP1
            or    Compressor(PState) = COMP2 ) and
      HP(PState) = OFF and
      LP(PState) = OFF and
      ELP(PState) = OFF and
      Comp1(PState) = OFF and
      Comp2(PState) = OFF and
      Isol1(PState) = OPEN and
      Isol2(PState) = OPEN and
      Vent1(PState) = OPEN and
      Vent2(PState) = OPEN and
      Bypass(PState) = OPEN and
      InOutlet(PState) = CLOSED and
      Comp1Alarm(PState) = OVERRIDE and
      Comp2Alarm(PState) = OVERRIDE )

 post Bypass(PState) = OPEN and
      Isol1(PState) = OPEN and
      Vent1(Pstate) = OPEN and
      Isol2(PState) = OPEN and
      Vent2(PState) = OPEN and
      InOutlet(PState) = CLOSED and
      Comp1(PState) = OFF and
      Comp2(PState) = OFF and
      EHP(PState) = UNKNOWN and
```

```
      HP(PState) = UNKNOWN and
      LP(PState) = UNKNOWN and
      ELP(PState) = UNKNOWN and
      Compressor(PState) = NONE and
      Comp1(PState) = OFF and
      Comp2(PState) = OFF and
      Comp1Lock(PState) = RESET and
      Comp2Lock(PState) = RESET and
      Comp1Alarm(PState) = UNKNOWN and
      Comp2Alarm(PState) = UNKNOWN and
      OperatorSwitch(PState) = UNKNOWN

  endpost
  endop
END
END
!
```

H.3. Animation Code Produced by the Animation Process.

H.3.1. Preamble and state initialisation.

The SIMSCRIPT code produced by the animator is as follows:

```
PREAMBLE


   TEMPORARY ENTITIES

      EVERY PlantState HAS
         A Bypass,
         A Isol1,
         A Isol2,
         A Vent1,
         A Vent2,
```

```
      A InOutlet,
      A EHP,
      A HP,
      A LP,
      A ELP,
      A Compressor,
      A Comp1,
      A Comp2,
      A Comp1Lock,
      A Comp2Lock,
      A Comp1Alarm,
      A Comp2Alarm,
      A OperatorSwitch

   DEFINE Bypass AS AN INTEGER VARIABLE
   DEFINE Isol1 AS AN INTEGER VARIABLE
   DEFINE Isol2 AS AN INTEGER VARIABLE
   DEFINE Vent1 AS AN INTEGER VARIABLE
   DEFINE Vent2 AS AN INTEGER VARIABLE
   DEFINE InOutlet AS AN INTEGER VARIABLE
   DEFINE EHP AS AN INTEGER VARIABLE
   DEFINE HP AS AN INTEGER VARIABLE
   DEFINE LP AS AN INTEGER VARIABLE
   DEFINE ELP AS AN INTEGER VARIABLE
   DEFINE Compressor AS AN INTEGER VARIABLE
   DEFINE Comp1 AS AN INTEGER VARIABLE
   DEFINE Comp2 AS AN INTEGER VARIABLE
   DEFINE Comp1Lock AS AN INTEGER VARIABLE
   DEFINE Comp2Lock AS AN INTEGER VARIABLE
   DEFINE Comp1Alarm AS AN INTEGER VARIABLE
   DEFINE Comp2Alarm AS AN INTEGER VARIABLE
   DEFINE OperatorSwitch AS AN INTEGER VARIABLE

DEFINE ..OPEN TO MEAN 1
DEFINE ..CLOSED TO MEAN 2
DEFINE ..ON TO MEAN 3
DEFINE ..OFF TO MEAN 4
```

```
      A InOutlet,
      A EHP,
      A HP,
      A LP,
      A ELP,
      A Compressor,
      A Comp1,
      A Comp2,
      A Comp1Lock,
      A Comp2Lock,
      A Comp1Alarm,
      A Comp2Alarm,
      A OperatorSwitch

   DEFINE Bypass AS AN INTEGER VARIABLE
   DEFINE Isol1 AS AN INTEGER VARIABLE
   DEFINE Isol2 AS AN INTEGER VARIABLE
   DEFINE Vent1 AS AN INTEGER VARIABLE
   DEFINE Vent2 AS AN INTEGER VARIABLE
   DEFINE InOutlet AS AN INTEGER VARIABLE
   DEFINE EHP AS AN INTEGER VARIABLE
   DEFINE HP AS AN INTEGER VARIABLE
   DEFINE LP AS AN INTEGER VARIABLE
   DEFINE ELP AS AN INTEGER VARIABLE
   DEFINE Compressor AS AN INTEGER VARIABLE
   DEFINE Comp1 AS AN INTEGER VARIABLE
   DEFINE Comp2 AS AN INTEGER VARIABLE
   DEFINE Comp1Lock AS AN INTEGER VARIABLE
   DEFINE Comp2Lock AS AN INTEGER VARIABLE
   DEFINE Comp1Alarm AS AN INTEGER VARIABLE
   DEFINE Comp2Alarm AS AN INTEGER VARIABLE
   DEFINE OperatorSwitch AS AN INTEGER VARIABLE

DEFINE ..OPEN TO MEAN 1
DEFINE ..CLOSED TO MEAN 2
DEFINE ..ON TO MEAN 3
DEFINE ..OFF TO MEAN 4
```

```
   DEFINE ..UNKNOWN TO MEAN 5
   DEFINE ..OVERRIDE TO MEAN 6
   DEFINE ..COMP1 TO MEAN 7
   DEFINE ..COMP2 TO MEAN 8
   DEFINE ..NONE TO MEAN 9
   DEFINE ..SET TO MEAN 10
   DEFINE ..RESET TO MEAN 11
   DEFINE PState AS A POINTER VARIABLE

END

ROUTINE Tim.Initialise.State

   CREATE A PlantState CALLED PState

      Bypass(PState) = ..OPEN
      Isol1(PState) = ..OPEN
      Isol2(PState) = ..OPEN
      Vent1(PState) = ..OPEN
      Vent2(PState) = ..OPEN
      InOutlet(PState) = ..CLOSED
      EHP(PState) = ..UNKNOWN
      HP(PState) = ..UNKNOWN
      LP(PState) = ..UNKNOWN
      ELP(PState) = ..UNKNOWN
      Compressor(PState) = ..NONE
      Comp1(PState) = ..OFF
      Comp2(PState) = ..OFF
      Comp1Lock(PState) = ..RESET
      Comp2Lock(PState) = ..RESET
      Comp1Alarm(PState) = ..UNKNOWN
      Comp2Alarm(PState) = ..UNKNOWN
      OperatorSwitch(PState) = ..UNKNOWN

END
```

H.3.2. Routine PlantSequencer.

```
ROUTINE PlantSequencer
   DEFINE Temp.Tim.Bypass.PState,
          Temp.Tim.Isol1.PState,
          Temp.Tim.Isol2.PState,
          Temp.Tim.Vent1.PState,
          Temp.Tim.Vent2.PState,
          Temp.Tim.InOutlet.PState,
          Temp.Tim.EHP.PState,
          Temp.Tim.HP.PState,
          Temp.Tim.LP.PState,
          Temp.Tim.ELP.PState,
          Temp.Tim.Compressor.PState,
          Temp.Tim.Comp1.PState,
          Temp.Tim.Comp2.PState,
          Temp.Tim.Comp1Lock.PState,
          Temp.Tim.Comp2Lock.PState,
          Temp.Tim.Comp1Alarm.PState,
          Temp.Tim.Comp2Alarm.PState,
          Temp.Tim.OperatorSwitch.PState
          AS INTEGER VARIABLES

   LET Temp.Tim.Bypass.PState = Bypass(PState)
   LET Temp.Tim.Isol1.PState = Isol1(PState)
   LET Temp.Tim.Isol2.PState = Isol2(PState)
   LET Temp.Tim.Vent1.PState = Vent1(PState)
   LET Temp.Tim.Vent2.PState = Vent2(PState)
   LET Temp.Tim.InOutlet.PState = InOutlet(PState)
   LET Temp.Tim.EHP.PState = EHP(PState)
   LET Temp.Tim.HP.PState = HP(PState)
   LET Temp.Tim.LP.PState = LP(PState)
   LET Temp.Tim.ELP.PState = ELP(PState)
   LET Temp.Tim.Compressor.PState = Compressor(PState)
   LET Temp.Tim.Comp1.PState = Comp1(PState)
   LET Temp.Tim.Comp2.PState = Comp2(PState)
   LET Temp.Tim.Comp1Lock.PState = Comp1Lock(PState)
   LET Temp.Tim.Comp2Lock.PState = Comp2Lock(PState)
   LET Temp.Tim.Comp1Alarm.PState = Comp1Alarm(PState)
```

```
   LET Temp.Tim.Comp2Alarm.PState = Comp2Alarm(PState)
    LET    Temp.Tim.OperatorSwitch.PState    =
OperatorSwitch(PState)

   CALL Tim.Update.Display GIVEN "PlantSequencer",
                                 "Pre "

   IF (    Bypass(PState) = ..OPEN
        and Isol1(PState) = ..OPEN
        and Isol2(PState) = ..OPEN
        and Vent1(PState) = ..OPEN
        and Vent2(PState) = ..OPEN
        and InOutlet(PState) = ..CLOSED
        and EHP(PState) = ..UNKNOWN
        and HP(PState) = ..UNKNOWN
        and LP(PState) = ..UNKNOWN
        and ELP(PState) = ..UNKNOWN
        and Compressor(PState) = ..NONE
        and Comp1(PState) = ..OFF
        and Comp2(PState) = ..OFF
        and Comp1Lock(PState) = ..RESET
        and Comp2Lock(PState) = ..RESET
        and Comp1Alarm(PState) = ..UNKNOWN
        and Comp2Alarm(PState) = ..UNKNOWN
        and OperatorSwitch(PState) = ..UNKNOWN )

      LET Bypass(PState) = ..OPEN
      LET Isol1(PState) = ..OPEN
      LET Vent1(Pstate) = ..OPEN
      LET Isol2(PState) = ..OPEN
      LET Vent2(PState) = ..OPEN
      LET InOutlet(PState) = ..CLOSED
      LET Comp1(PState) = ..OFF
      LET Comp2(PState) = ..OFF
      LET EHP(PState) = ..UNKNOWN
      LET HP(PState) = ..UNKNOWN
      LET LP(PState) = ..UNKNOWN
```

```
      LET ELP(PState) = ..UNKNOWN
      LET Compressor(PState) = ..NONE
      LET Comp1(PState) = ..OFF
      LET Comp2(PState) = ..OFF
      LET Comp1Lock(PState) = ..RESET
      LET Comp2Lock(PState) = ..RESET
      LET Comp1Alarm(PState) = ..UNKNOWN
      LET Comp2Alarm(PState) = ..UNKNOWN
      LET OperatorSwitch(PState) = ..UNKNOWN
   ENDIF
   CALL Tim.Update.Display GIVEN "PlantSequencer",
                                 "Post"

DECOMPOSITION

     CALL StartUpPlant
     CALL DoWorkingCycle
     CALL ShutPlantDown
END
```

H.3.3. Routine StartUpPlant.

```
ROUTINE StartUpPlant
   DEFINE Temp.Tim.Bypass.PState,
          Temp.Tim.Isol1.PState,
          Temp.Tim.Isol2.PState,
          Temp.Tim.Vent1.PState,
          Temp.Tim.Vent2.PState,
          Temp.Tim.InOutlet.PState,
          Temp.Tim.EHP.PState,
          Temp.Tim.HP.PState,
          Temp.Tim.LP.PState,
          Temp.Tim.ELP.PState,
          Temp.Tim.Compressor.PState,
          Temp.Tim.Comp1.PState,
          Temp.Tim.Comp2.PState,
          Temp.Tim.Comp1Lock.PState,
```

```
            Temp.Tim.Comp2Lock.PState,
            Temp.Tim.Comp1Alarm.PState,
            Temp.Tim.Comp2Alarm.PState,
            Temp.Tim.OperatorSwitch.PState
            AS INTEGER VARIABLES

   LET Temp.Tim.Bypass.PState = Bypass(PState)
   LET Temp.Tim.Isol1.PState = Isol1(PState)
   LET Temp.Tim.Isol2.PState = Isol2(PState)
   LET Temp.Tim.Vent1.PState = Vent1(PState)
   LET Temp.Tim.Vent2.PState = Vent2(PState)
   LET Temp.Tim.InOutlet.PState = InOutlet(PState)
   LET Temp.Tim.EHP.PState = EHP(PState)
   LET Temp.Tim.HP.PState = HP(PState)
   LET Temp.Tim.LP.PState = LP(PState)
   LET Temp.Tim.ELP.PState = ELP(PState)
   LET Temp.Tim.Compressor.PState = Compressor(PState)
   LET Temp.Tim.Comp1.PState = Comp1(PState)
   LET Temp.Tim.Comp2.PState = Comp2(PState)
   LET Temp.Tim.Comp1Lock.PState = Comp1Lock(PState)
   LET Temp.Tim.Comp2Lock.PState = Comp2Lock(PState)
   LET Temp.Tim.Comp1Alarm.PState = Comp1Alarm(PState)
   LET Temp.Tim.Comp2Alarm.PState = Comp2Alarm(PState)
    LET   Temp.Tim.OperatorSwitch.PState   =
OperatorSwitch(PState)

   CALL Tim.Update.Display GIVEN "StartUpPlant",
                                 "Pre "

   IF (    Bypass(PState) = ..OPEN
       and Isol1(PState) = ..OPEN
       and Isol2(PState) = ..OPEN
       and Vent1(PState) = ..OPEN
       and Vent2(PState) = ..OPEN
       and InOutlet(PState) = ..CLOSED
       and EHP(PState) = ..UNKNOWN
       and HP(PState) = ..UNKNOWN
```

```
    and LP(PState) = ..UNKNOWN
    and ELP(PState) = ..UNKNOWN
    and Compressor(PState) = ..NONE
    and Comp1(PState) = ..OFF
    and Comp2(PState) = ..OFF
    and Comp1Lock(PState) = ..RESET
    and Comp2Lock(PState) = ..RESET
    and Comp1Alarm(PState) = ..UNKNOWN
    and Comp2Alarm(PState) = ..UNKNOWN
    and OperatorSwitch(PState) = ..UNKNOWN )

SELECT CASE .....
   CASE ...

        LET Compressor(PState) = ..COMP1
        LET Comp1Lock(PState) = ..RESET
        LET Comp1(PState) = ..ON
        LET Isol1(PState) = ..OPEN
        LET Vent1(PState) = ..CLOSED
        LET Bypass(PState) = ..CLOSED
        LET InOutlet(PState) = ..OPEN
        LET EHP(PState) = ..OFF
        LET HP(PState) = ..OFF
        LET LP(PState) = ..OFF
        LET ELP(PState) = ..OFF
        LET Comp1Alarm(PState) = ..OFF
        LET Comp2Lock(PState) = ..RESET
        LET Comp2(PState) = ..OFF
        LET Isol2(PState) = ..CLOSED
        LET Vent2(PState) = ..OPEN
        LET Comp2Alarm(PState) = ..OVERRIDE
        LET OperatorSwitch(PState) = ..ON
   CASE ...

        LET Compressor(PState) = ..COMP2
        LET Comp2Lock(PState) = ..RESET
        LET Comp2(PState) = ..ON
```

```
            LET Isol2(PState) = ..OPEN
            LET Vent2(PState) = ..CLOSED
            LET Bypass(PState) = ..CLOSED
            LET InOutlet(PState) = ..OPEN
            LET EHP(PState) = ..OFF
            LET HP(PState) = ..OFF
            LET LP(PState) = ..OFF
            LET ELP(PState) = ..OFF
            LET Comp2Alarm(PState) = ..OFF
            LET Comp1Lock(PState) = ..SET
            LET Comp1(PState) = ..OFF
            LET Isol1(PState) = ..CLOSED
            LET Vent1(PState) = ..OPEN
            LET Comp1Alarm(PState) = ..OVERRIDE
            LET OperatorSwitch(PState) = ..ON
    CASE ...

            LET Compressor(PState) = ..NONE
            LET Comp1Lock(PState) = ..SET
            LET Comp2Lock(PState) = ..SET
            LET Comp1(PState) = ..OFF
            LET Comp2(PState) = ..OFF
            LET Isol1(PState) = ..OPEN
            LET Vent1(PState) = ..OPEN
            LET Isol2(PState) = ..OPEN
            LET Vent2(PState) = ..OPEN
            LET Bypass(PState) = ..OPEN
            LET InOutlet(PState) = ..CLOSED
            LET EHP(PState) = ..OFF
            LET HP(PState) = ..OFF
            LET LP(PState) = ..OFF
            LET ELP(PState) = ..OFF
            LET Comp1Alarm(PState) = ..OVERRIDE
            LET Comp2Alarm(PState) = ..OVERRIDE
            LET OperatorSwitch(PState) = ..ON
ENDSELECT
ENDIF
```

```
   CALL Tim.Update.Display GIVEN "StartUpPlant",
                                 "Post"

END
```

H.3.4. Routine DoWorkingCycle.

```
ROUTINE DoWorkingCycle
   DEFINE Temp.Tim.Bypass.PState,
          Temp.Tim.Isol1.PState,
          Temp.Tim.Isol2.PState,
          Temp.Tim.Vent1.PState,
          Temp.Tim.Vent2.PState,
          Temp.Tim.InOutlet.PState,
          Temp.Tim.EHP.PState,
          Temp.Tim.HP.PState,
          Temp.Tim.LP.PState,
          Temp.Tim.ELP.PState,
          Temp.Tim.Compressor.PState,
          Temp.Tim.Comp1.PState,
          Temp.Tim.Comp2.PState,
          Temp.Tim.Comp1Lock.PState,
          Temp.Tim.Comp2Lock.PState,
          Temp.Tim.Comp1Alarm.PState,
          Temp.Tim.Comp2Alarm.PState,
          Temp.Tim.OperatorSwitch.PState
          AS INTEGER VARIABLES

   LET Temp.Tim.Bypass.PState = Bypass(PState)
   LET Temp.Tim.Isol1.PState = Isol1(PState)
   LET Temp.Tim.Isol2.PState = Isol2(PState)
   LET Temp.Tim.Vent1.PState = Vent1(PState)
   LET Temp.Tim.Vent2.PState = Vent2(PState)
   LET Temp.Tim.InOutlet.PState = InOutlet(PState)
   LET Temp.Tim.EHP.PState = EHP(PState)
   LET Temp.Tim.HP.PState = HP(PState)
   LET Temp.Tim.LP.PState = LP(PState)
```

```
   LET Temp.Tim.ELP.PState = ELP(PState)
   LET Temp.Tim.Compressor.PState = Compressor(PState)
   LET Temp.Tim.Comp1.PState = Comp1(PState)
   LET Temp.Tim.Comp2.PState = Comp2(PState)
   LET Temp.Tim.Comp1Lock.PState = Comp1Lock(PState)
   LET Temp.Tim.Comp2Lock.PState = Comp2Lock(PState)
   LET Temp.Tim.Comp1Alarm.PState = Comp1Alarm(PState)
   LET Temp.Tim.Comp2Alarm.PState = Comp2Alarm(PState)
    LET    Temp.Tim.OperatorSwitch.PState    =
OperatorSwitch(PState)

   CALL Tim.Update.Display GIVEN "DoWorkingCycle",
                                 "Pre "

   IF (     (    Compressor(PState) = ..COMP1
            and Comp1Lock(PState) = ..RESET
            and Comp1(PState) = ..ON
            and Isol1(PState) = ..OPEN
            and Vent1(PState) = ..CLOSED
            and Bypass(PState) = ..CLOSED
            and InOutlet(PState) = ..OPEN
            and EHP(PState) = ..OFF
            and HP(PState) = ..OFF
            and LP(PState) = ..OFF
            and ELP(PState) = ..OFF
            and Comp1Alarm(PState) = ..OFF
            and Comp2Lock(PState) = ..RESET
            and Comp2(PState) = ..OFF
            and Isol2(PState) = ..CLOSED
            and Vent2(PState) = ..OPEN
            and Comp2Alarm(PState) = ..OVERRIDE
            and OperatorSwitch(PState) = ..ON )
         OR (    Compressor(PState) = ..COMP2
            and Comp2Lock(PState) = ..RESET
            and Comp2(PState) = ..ON
            and Isol2(PState) = ..OPEN
            and Vent2(PState) = ..CLOSED
```

```
        and Bypass(PState) = ..CLOSED
        and InOutlet(PState) = ..OPEN
        and EHP(PState) = ..OFF
        and HP(PState) = ..OFF
        and LP(PState) = ..OFF
        and ELP(PState) = ..OFF
        and Comp2Alarm(PState) = ..OFF
        and Comp1Lock(PState) = ..SET
        and Comp1(PState) = ..OFF
        and Isol1(PState) = ..CLOSED
        and Vent1(PState) = ..OPEN
        and Comp1Alarm(PState) = ..OVERRIDE
        and OperatorSwitch(PState) = ..ON )
    OR (    Compressor(PState) = ..NONE
        and Comp1Lock(PState) = ..SET
        and Comp2Lock(PState) = ..SET
        and Comp1(PState) = ..OFF
        and Comp2(PState) = ..OFF
        and Isol1(PState) = ..OPEN
        and Vent1(PState) = ..OPEN
        and Isol2(PState) = ..OPEN
        and Vent2(PState) = ..OPEN
        and Bypass(PState) = ..OPEN
        and InOutlet(PState) = ..CLOSED
        and EHP(PState) = ..OFF
        and HP(PState) = ..OFF
        and LP(PState) = ..OFF
        and ELP(PState) = ..OFF
        and Comp1Alarm(PState) = ..OVERRIDE
        and Comp2Alarm(PState) = ..OVERRIDE
        and OperatorSwitch(PState) = ..ON ) )

IF (    Temp.Tim.Compressor.PState = ..NONE )
    LET Bypass(PState) = Temp.Tim.Bypass.PState
    LET Isol1(PState) = Temp.Tim.Isol1.PState
    LET Isol2(PState) = Temp.Tim.Isol2.PState
    LET Vent1(PState) = Temp.Tim.Vent1.PState
```

```
          LET Vent2(PState) = Temp.Tim.Vent2.PState
          LET InOutlet(PState) = Temp.Tim.InOutlet.PState
          LET EHP(PState) = Temp.Tim.EHP.PState
          LET HP(PState) = Temp.Tim.HP.PState
          LET LP(PState) = Temp.Tim.LP.PState
          LET ELP(PState) = Temp.Tim.ELP.PState
                LET Compressor(PState) =
Temp.Tim.Compressor.PState
          LET Comp1(PState) = Temp.Tim.Comp1.PState
          LET Comp2(PState) = Temp.Tim.Comp2.PState
          LET Comp1Lock(PState) = Temp.Tim.Comp1Lock.PState
          LET Comp2Lock(PState) = Temp.Tim.Comp2Lock.PState
                LET Comp1Alarm(PState) =
Temp.Tim.Comp1Alarm.PState
                LET Comp2Alarm(PState) =
Temp.Tim.Comp2Alarm.PState
        LET OperatorSwitch(PState) =
                          Temp.Tim.OperatorSwitch.PState
      ENDIF

      IF (   Temp.Tim.Compressor.PState = ..COMP1
          OR Temp.Tim.Compressor.PState = ..COMP2 )
          SELECT CASE .....
             CASE ...

                   LET OperatorSwitch(PState) = ..OFF
                   LET EHP(PState) = ..OFF
                LET Compressor(PState) =
                         Temp.Tim.Compressor.PState
             CASE ...

                   LET EHP(PState) = ..ON
        LET OperatorSwitch(PState) =
                          Temp.Tim.OperatorSwitch.PState
                 LET Compressor(PState) =
Temp.Tim.Compressor.PState
             CASE ...
```

```
                    LET Compressor(PState) = ..NONE
                    LET EHP(PState) = ..OFF
          LET OperatorSwitch(PState) =
                         Temp.Tim.OperatorSwitch.PState
                    LET Comp1Lock(PState) = ..SET
                    LET Comp2Lock(PState) = ..SET
            ENDSELECT
            LET HP(PState) = ..OFF
            LET LP(PState) = ..OFF
            LET ELP(PState) = ..OFF
            LET Comp1(PState) = ..OFF
            LET Comp2(PState) = ..OFF
            LET Isol1(PState) = ..OPEN
            LET Isol2(PState) = ..OPEN
            LET Vent1(PState) = ..OPEN
            LET Vent2(PState) = ..OPEN
            LET Bypass(PState) = ..OPEN
            LET InOutlet(PState) = ..CLOSED
            LET Comp1Alarm(PState) = ..OVERRIDE
            LET Comp2Alarm(PState) = ..OVERRIDE
         ENDIF

      ENDIF
      CALL Tim.Update.Display GIVEN "DoWorkingCycle",
                                    "Post"

END
```

H.3.5. Routine ShutPlantDown.

```
ROUTINE ShutPlantDown
   DEFINE Temp.Tim.Bypass.PState,
          Temp.Tim.Isol1.PState,
          Temp.Tim.Isol2.PState,
          Temp.Tim.Vent1.PState,
          Temp.Tim.Vent2.PState,
          Temp.Tim.InOutlet.PState,
```

```
            Temp.Tim.EHP.PState,
            Temp.Tim.HP.PState,
            Temp.Tim.LP.PState,
            Temp.Tim.ELP.PState,
            Temp.Tim.Compressor.PState,
            Temp.Tim.Comp1.PState,
            Temp.Tim.Comp2.PState,
            Temp.Tim.Comp1Lock.PState,
            Temp.Tim.Comp2Lock.PState,
            Temp.Tim.Comp1Alarm.PState,
            Temp.Tim.Comp2Alarm.PState,
            Temp.Tim.OperatorSwitch.PState
            AS INTEGER VARIABLES

   LET Temp.Tim.Bypass.PState = Bypass(PState)
   LET Temp.Tim.Isol1.PState = Isol1(PState)
   LET Temp.Tim.Isol2.PState = Isol2(PState)
   LET Temp.Tim.Vent1.PState = Vent1(PState)
   LET Temp.Tim.Vent2.PState = Vent2(PState)
   LET Temp.Tim.InOutlet.PState = InOutlet(PState)
   LET Temp.Tim.EHP.PState = EHP(PState)
   LET Temp.Tim.HP.PState = HP(PState)
   LET Temp.Tim.LP.PState = LP(PState)
   LET Temp.Tim.ELP.PState = ELP(PState)
   LET Temp.Tim.Compressor.PState = Compressor(PState)
   LET Temp.Tim.Comp1.PState = Comp1(PState)
   LET Temp.Tim.Comp2.PState = Comp2(PState)
   LET Temp.Tim.Comp1Lock.PState = Comp1Lock(PState)
   LET Temp.Tim.Comp2Lock.PState = Comp2Lock(PState)
   LET Temp.Tim.Comp1Alarm.PState = Comp1Alarm(PState)
   LET Temp.Tim.Comp2Alarm.PState = Comp2Alarm(PState)
    LET   Temp.Tim.OperatorSwitch.PState   =
OperatorSwitch(PState)

   CALL Tim.Update.Display GIVEN "ShutPlantDown",
                                 "Pre "
```

```
IF (     (     Compressor(PState) = ..NONE
           and Comp1Lock(PState) = ..SET
           and Comp2Lock(PState) = ..SET
           and Comp1(PState) = ..OFF
           and Comp2(PState) = ..OFF
           and Isol1(PState) = ..OPEN
           and Vent1(PState) = ..OPEN
           and Isol2(PState) = ..OPEN
           and Vent2(PState) = ..OPEN
           and Bypass(PState) = ..OPEN
           and InOutlet(PState) = ..CLOSED
           and EHP(PState) = ..OFF
           and HP(PState) = ..OFF
           and LP(PState) = ..OFF
           and ELP(PState) = ..OFF
           and Comp1Alarm(PState) = ..OVERRIDE
           and Comp2Alarm(PState) = ..OVERRIDE
           and OperatorSwitch(PState) = ..ON )
         OR (     (     (     OperatorSwitch(PState) = ..OFF
                          and EHP(PState) = ..OFF )
                    OR (     EHP(PState) = ..ON
                          and OperatorSwitch(PState) = ..ON )
)
           and (     Compressor(PState) = ..COMP1
                 OR Compressor(PState) = ..COMP2 )
           and HP(PState) = ..OFF
           and LP(PState) = ..OFF
           and ELP(PState) = ..OFF
           and Comp1(PState) = ..OFF
           and Comp2(PState) = ..OFF
           and Isol1(PState) = ..OPEN
           and Isol2(PState) = ..OPEN
           and Vent1(PState) = ..OPEN
           and Vent2(PState) = ..OPEN
           and Bypass(PState) = ..OPEN
           and InOutlet(PState) = ..CLOSED
           and Comp1Alarm(PState) = ..OVERRIDE
```

```
            and Comp2Alarm(PState) = ..OVERRIDE ) )

      LET Bypass(PState) = ..OPEN
      LET Isol1(PState) = ..OPEN
      LET Vent1(Pstate) = ..OPEN
      LET Isol2(PState) = ..OPEN
      LET Vent2(PState) = ..OPEN
      LET InOutlet(PState) = ..CLOSED
      LET Comp1(PState) = ..OFF
      LET Comp2(PState) = ..OFF
      LET EHP(PState) = ..UNKNOWN
      LET HP(PState) = ..UNKNOWN
      LET LP(PState) = ..UNKNOWN
      LET ELP(PState) = ..UNKNOWN
      LET Compressor(PState) = ..NONE
      LET Comp1(PState) = ..OFF
      LET Comp2(PState) = ..OFF
      LET Comp1Lock(PState) = ..RESET
      LET Comp2Lock(PState) = ..RESET
      LET Comp1Alarm(PState) = ..UNKNOWN
      LET Comp2Alarm(PState) = ..UNKNOWN
      LET OperatorSwitch(PState) = ..UNKNOWN
   ENDIF
   CALL Tim.Update.Display GIVEN "ShutPlantDown",
                                 "Post"

END

MAIN

END
```

H.3.6. Routine Tim.Update.Display.

```
ROUTINE Tim.Update.Display GIVEN Op.Name, Condition

   DEFINE Op.Name, Condition AS TEXT VARIABLES
```

```
   LET DTVAL.A(DFIELD.F( ** Op Box Name ** , ** Main.Form
** )
            = Op.Name
   DISPLAY DFIELD.F( ** Op Box Name ** , ** Main.Form ** )
   LET  DTVAL.A(DFIELD.F(**  Condition  Box  Name  **,**
Main.Form **)
            = Condition
   DISPLAY  DFIELD.F(  **  Condition  Box  Name  **  ,  **
Main.Form **)

   SELECT CASE Bypass( PState )

      CASE ..OPEN
         ** Update appropriate icon **
      CASE ..CLOSED
         ** Update appropriate icon **

   ENDSELECT


   SELECT CASE Isol1( PState )

      CASE ..OPEN
         ** Update appropriate icon **

      CASE ..CLOSED
         ** Update appropriate icon **

   ENDSELECT


   SELECT CASE Isol2( PState )

      CASE ..OPEN
         ** Update appropriate icon **
      CASE ..CLOSED
         ** Update appropriate icon **
```

```
ENDSELECT


SELECT CASE Vent1( PState )

   CASE ..OPEN
      ** Update appropriate icon **
   CASE ..CLOSED
      ** Update appropriate icon **

ENDSELECT


SELECT CASE Vent2( PState )

   CASE ..OPEN
      ** Update appropriate icon **
   CASE ..CLOSED
      ** Update appropriate icon **

ENDSELECT


SELECT CASE InOutlet( PState )

   CASE ..OPEN
      ** Update appropriate icon **
   CASE ..CLOSED
      ** Update appropriate icon **

ENDSELECT


SELECT CASE EHP( PState )

   CASE ..ON
      ** Update appropriate icon **
```

```
   CASE ..OFF
      ** Update appropriate icon **
   CASE ..UNKNOWN
      ** Update appropriate icon **

ENDSELECT


SELECT CASE HP( PState )

   CASE ..ON
      ** Update appropriate icon **
   CASE ..OFF
      ** Update appropriate icon **
   CASE ..UNKNOWN
      ** Update appropriate icon **

ENDSELECT


SELECT CASE LP( PState )

   CASE ..ON
      ** Update appropriate icon **
   CASE ..OFF
      ** Update appropriate icon **
   CASE ..UNKNOWN
      ** Update appropriate icon **
   CASE ..OVERRIDE
      ** Update appropriate icon **

ENDSELECT


SELECT CASE ELP( PState )

   CASE ..ON
```

```
      ** Update appropriate icon **
   CASE ..OFF
      ** Update appropriate icon **
   CASE ..UNKNOWN
      ** Update appropriate icon **
   CASE ..OVERRIDE
      ** Update appropriate icon **

ENDSELECT


SELECT CASE Compressor( PState )

   CASE ..COMP1
      ** Update appropriate icon **
   CASE ..COMP2
      ** Update appropriate icon **
   CASE ..NONE
      ** Update appropriate icon **

ENDSELECT


SELECT CASE Comp1( PState )

   CASE ..ON
      ** Update appropriate icon **
   CASE ..OFF
      ** Update appropriate icon **

ENDSELECT


SELECT CASE Comp2( PState )

   CASE ..ON
      ** Update appropriate icon **
```

```
   CASE ..OFF
      ** Update appropriate icon **

ENDSELECT


SELECT CASE Comp1Lock( PState )

   CASE ..SET
      ** Update appropriate icon **
   CASE ..RESET
      ** Update appropriate icon **

ENDSELECT


SELECT CASE Comp2Lock( PState )

   CASE ..SET
      ** Update appropriate icon **
   CASE ..RESET
      ** Update appropriate icon **

ENDSELECT


SELECT CASE Comp1Alarm( PState )

   CASE ..ON
      ** Update appropriate icon **
   CASE ..OFF
      ** Update appropriate icon **
   CASE ..UNKNOWN
      ** Update appropriate icon **
   CASE ..OVERRIDE
      ** Update appropriate icon **
```

```
   ENDSELECT


   SELECT CASE Comp2Alarm( PState )

      CASE ..ON
         ** Update appropriate icon **
      CASE ..OFF
         ** Update appropriate icon **
      CASE ..UNKNOWN
         ** Update appropriate icon **
      CASE ..OVERRIDE
         ** Update appropriate icon **

   ENDSELECT


   SELECT CASE OperatorSwitch( PState )

      CASE ..ON
         ** Update appropriate icon **
      CASE ..OFF
         ** Update appropriate icon **
      CASE ..UNKNOWN
         ** Update appropriate icon **

   ENDSELECT

END
```

H.3.7. Adding animated graphical displays.

Note that in StartUp and DoWorkingCycle there are incomplete SELECT CASE statements. These are used in the translation process when a number of possible solutions to the post condition exist. At present it is up to the developer to add a method of selecting which one is to be demonstrated. Also in the update graphics routines there are blanks to be filled in detailing the effects on the display of a certain value of a variable.

The following figures (Figure 28 to Figure 37) show displays from the animated prototype of this specification.

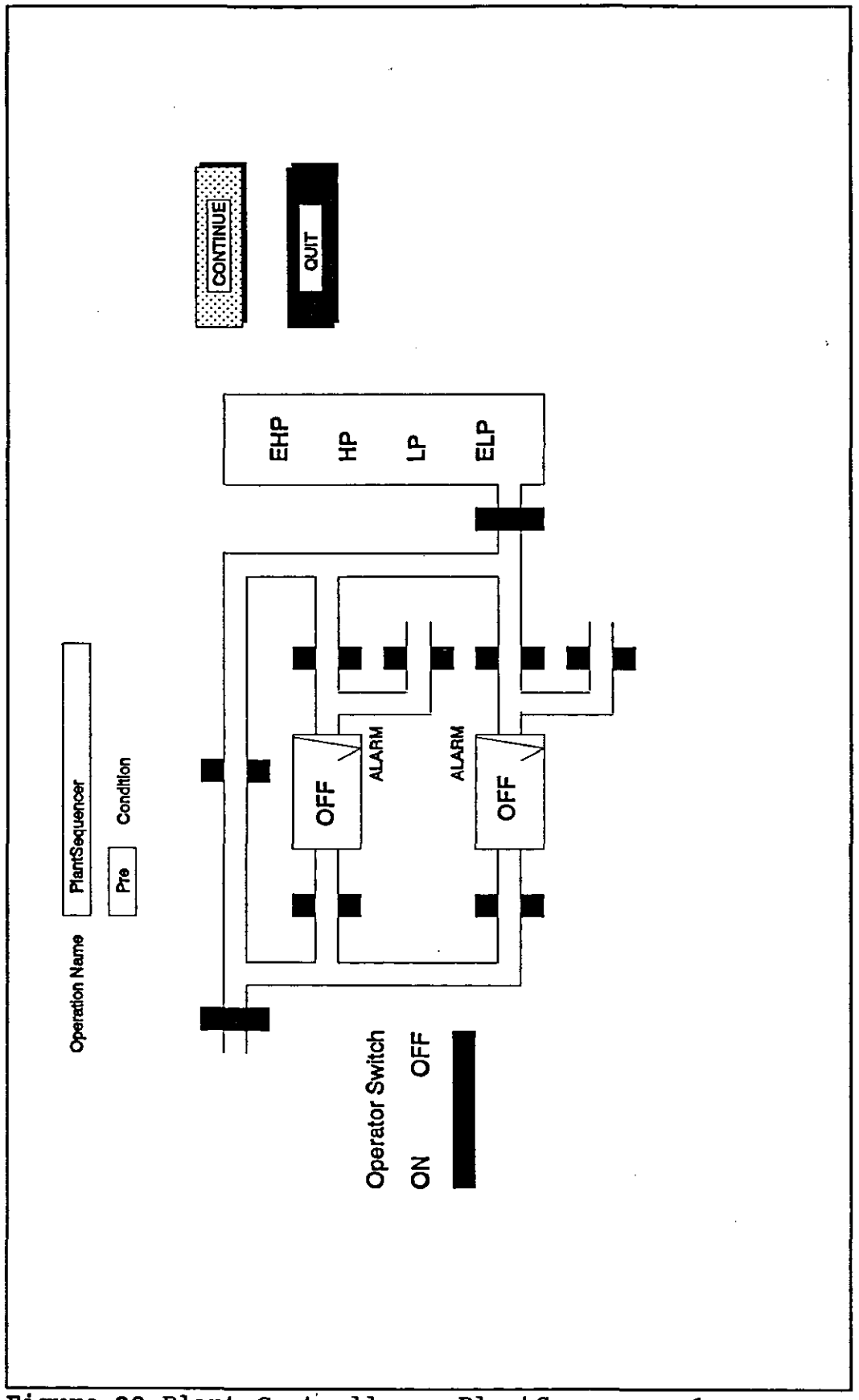


Figure 28 Plant Controller - PlantSequencer 1

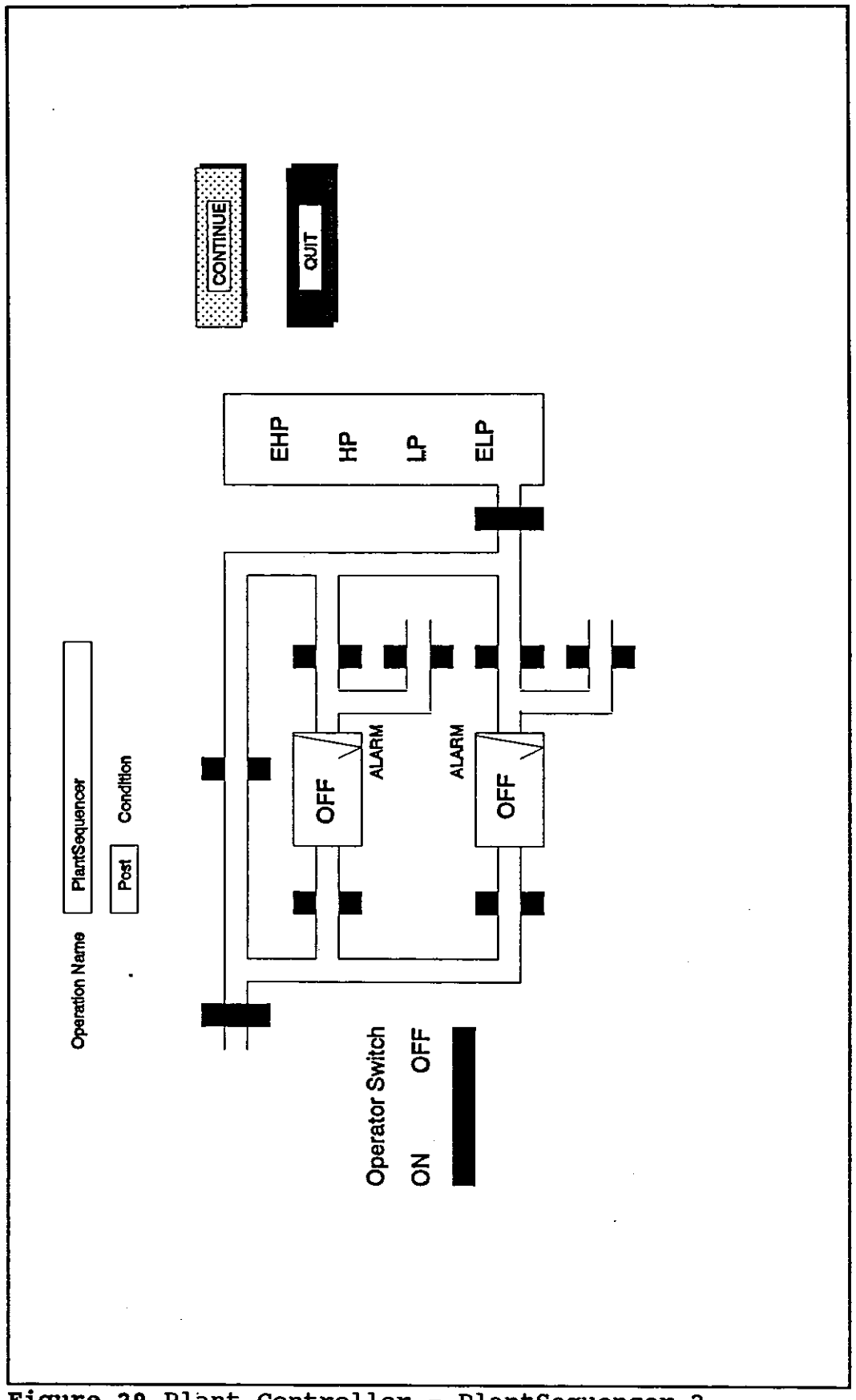


Figure 29 Plant Controller - PlantSequencer 2

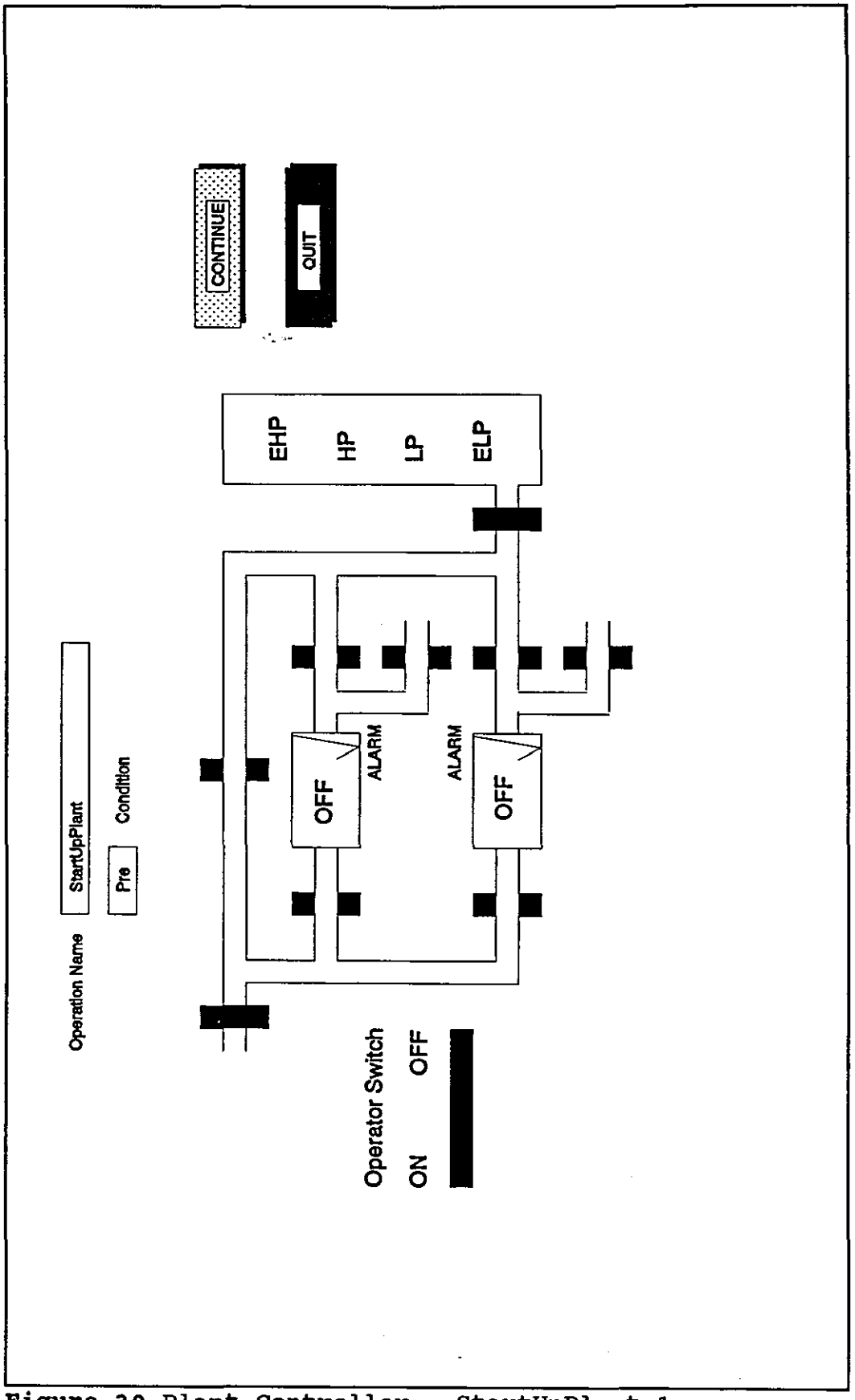


Figure 30 Plant Controller - StartUpPlant 1

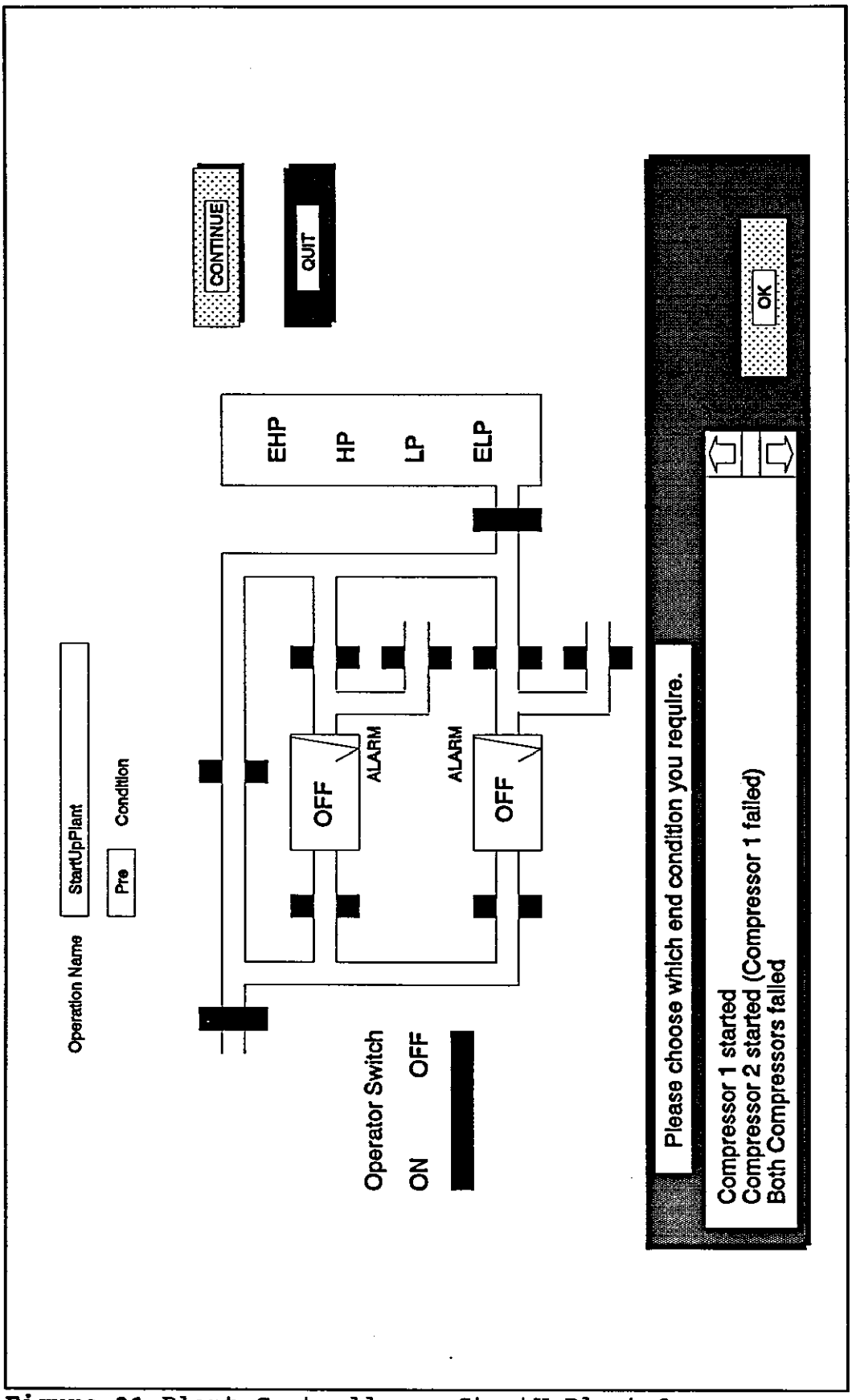


Figure 31 Plant Controller - StartUpPlant 2

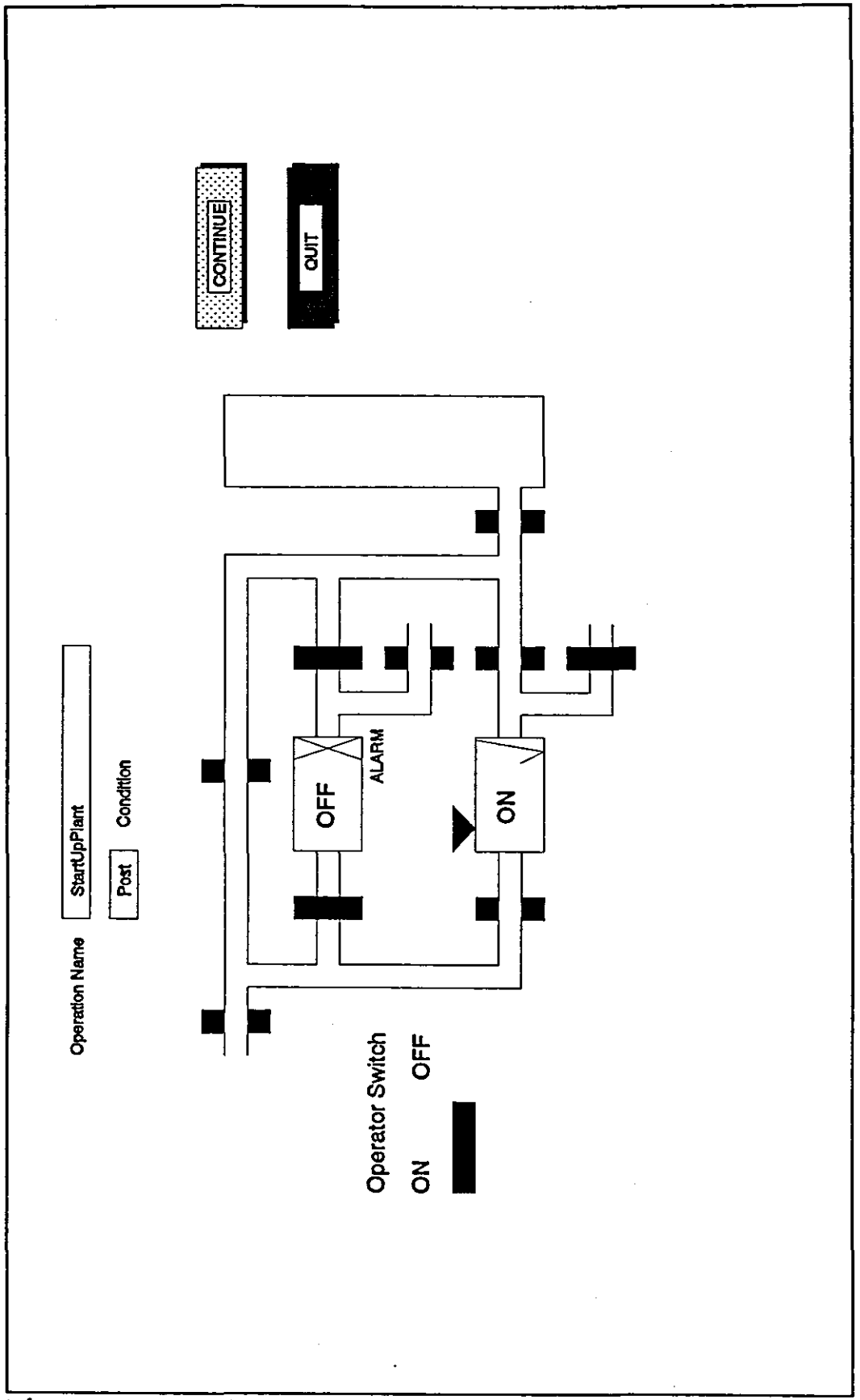


Figure 32 Plant Controller - StartUpPlant 3

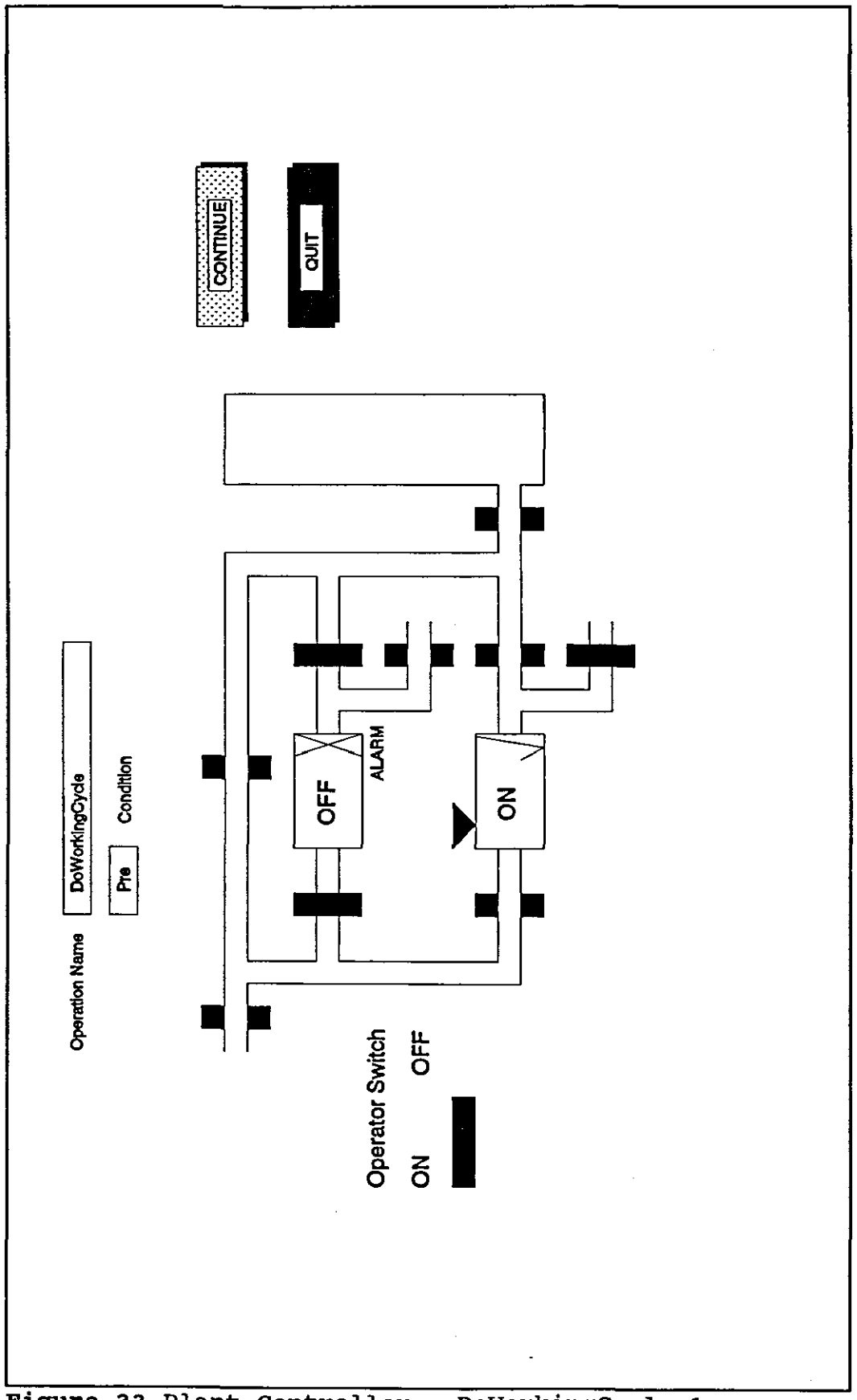


Figure 33 Plant Controller - DoWorkingCycle 1

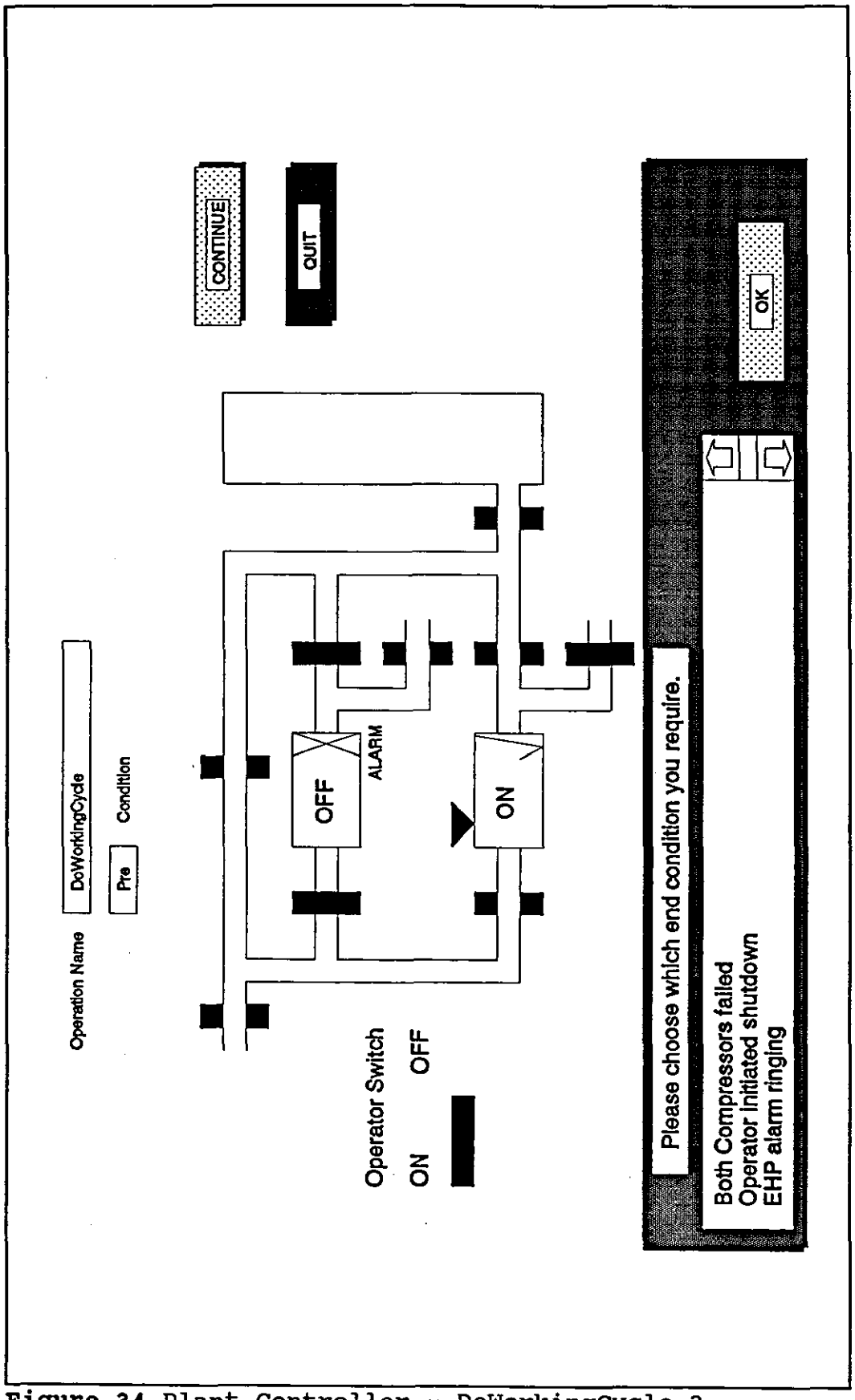


Figure 34 Plant Controller - DoWorkingCycle 2

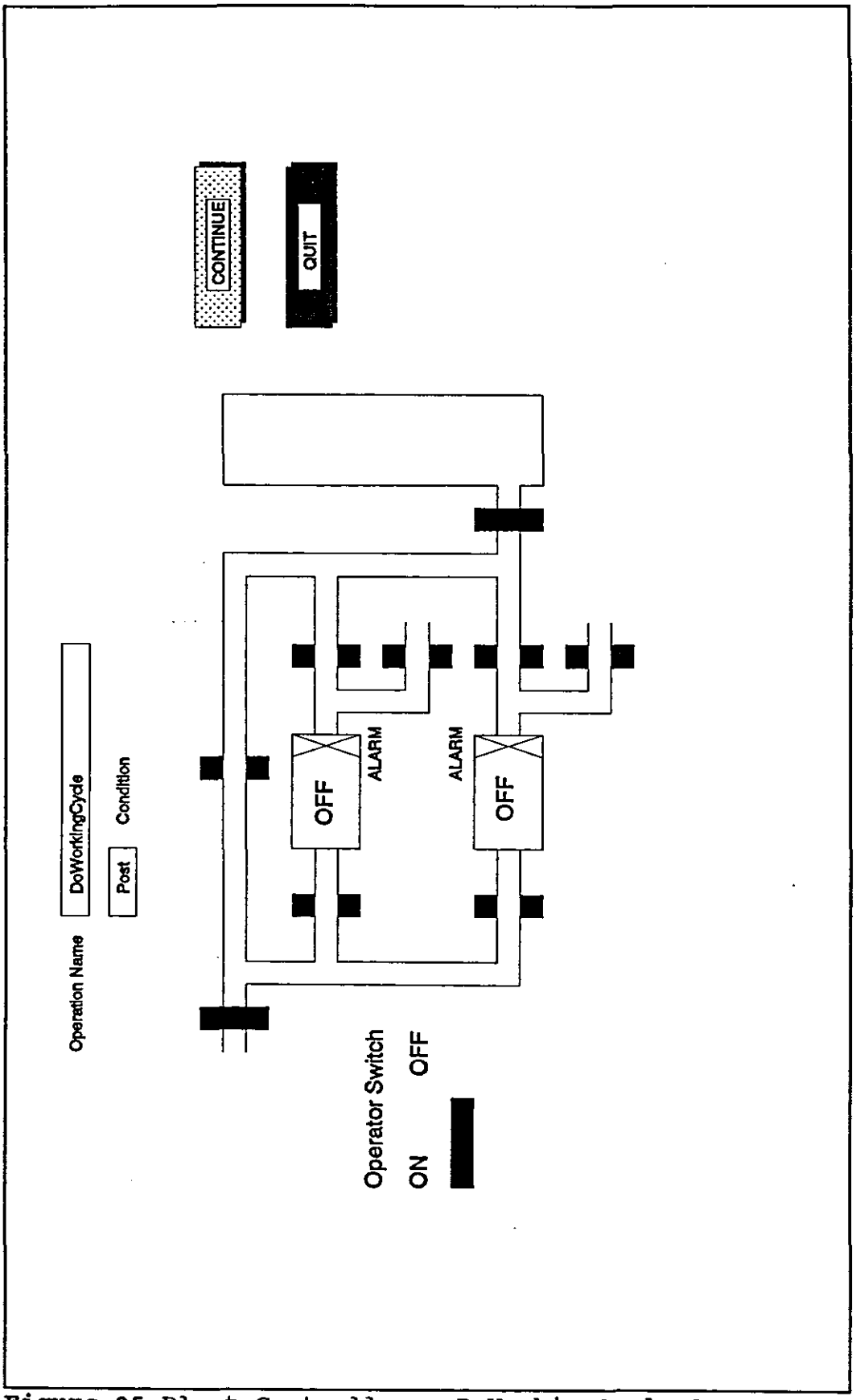


Figure 35 Plant Controller - DoWorkingCycle 3

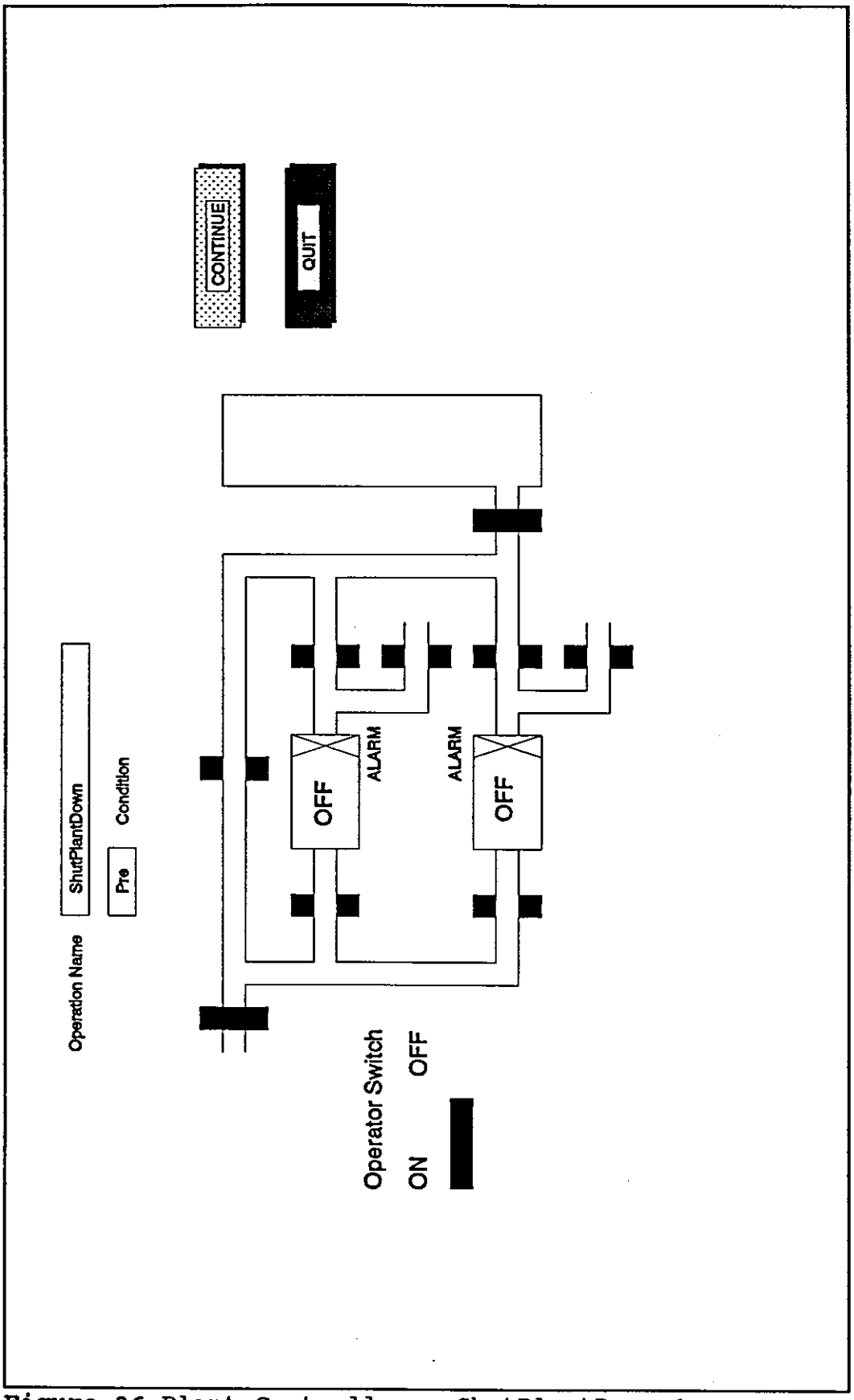


Figure 36 Plant Controller - ShutPlantDown 1

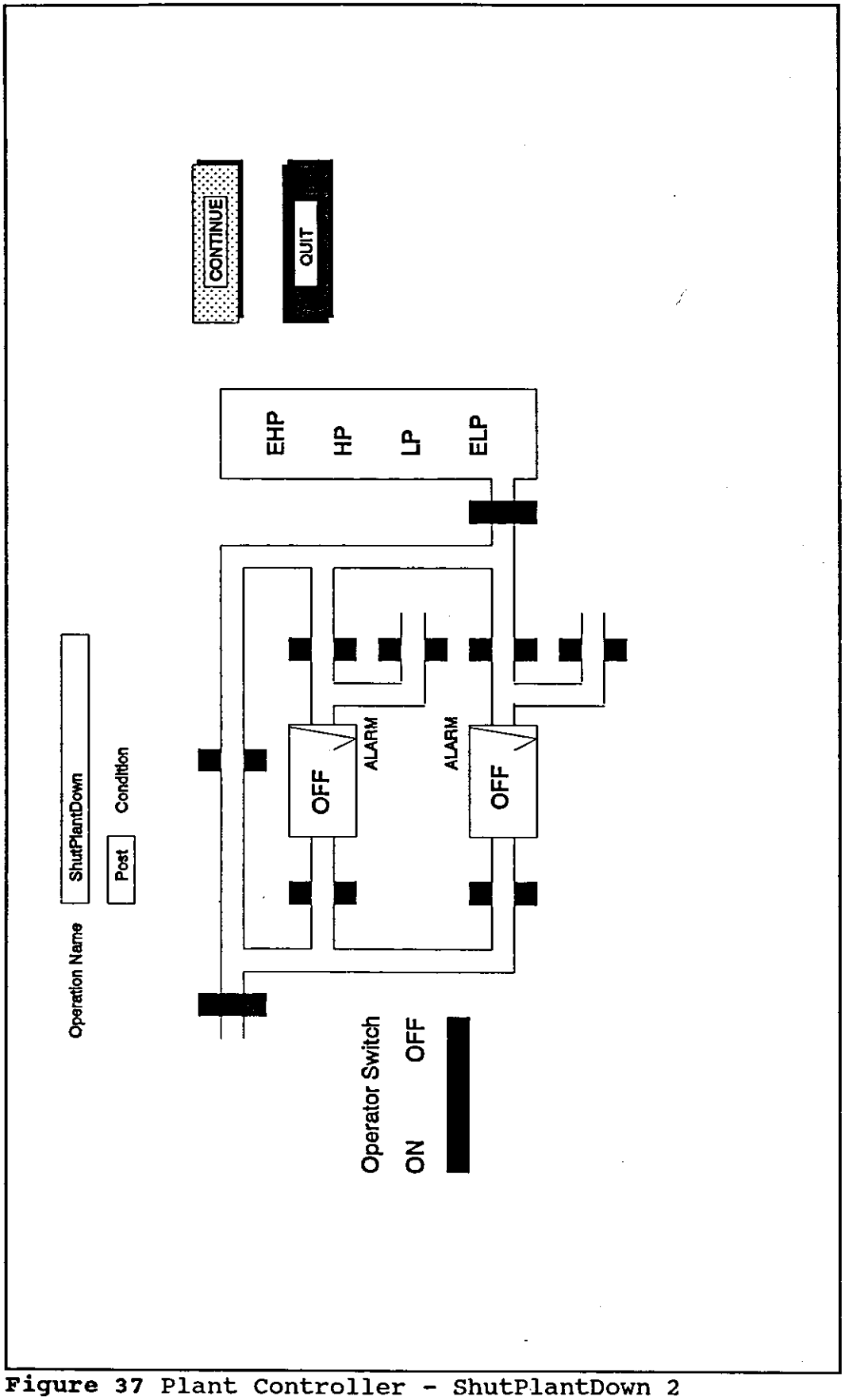


Figure 37 Plant Controller - ShutPlantDown 2

H.3.8. Further decomposition of the specification.

The specification of the plant sequencer was further refined by decomposing the DoWorkingCycle operation into five sub-operations: WaitForCycleStart, StartCompressor, MonitorPlantAndCompressor, StopCompressor, PrepareNextCompressor as shown in Figure 38 below.

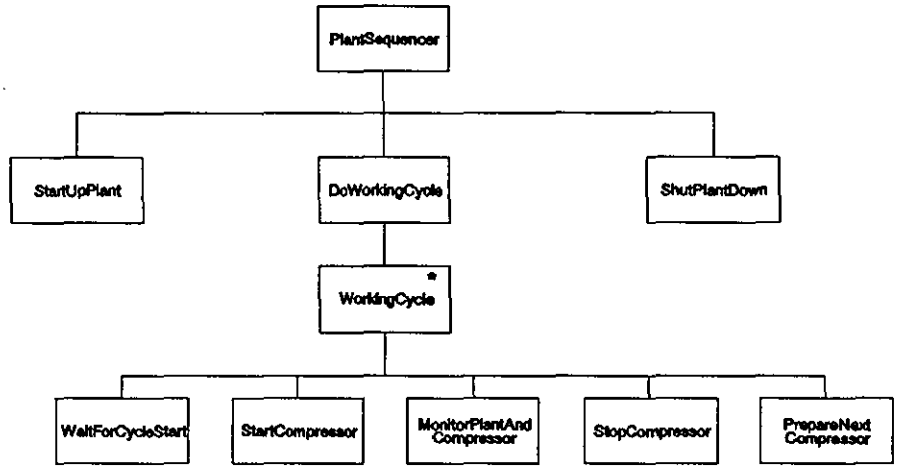


Figure 38 Further Decomposition of the Plant Controller Specification.