Correctly specifying the requirements of a software system is a crucial task; mistakes made here affect all subsequent stages of software development. Experience has shown that rectifying such errors is both costly and time consuming [1]. As a result, many techniques have been developed to aid the specification process: De-Marco methods, CORE, SSADM for instance [2]. 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 [3].

Much of the driving force for this has come from academia. By contrast, industry has shown considerable resistance to the use of formal methods. The most common doubt is the ability to scale-up the formal specification process. At the moment, most examples of such methods involve small systems only. How well will they work when applied to large projects? This is a genuine concern which industry is well aware of. But there is also another major problem which few seem to recognise. That is, having formally specified a system, how can you confirm that the specification defines precisely what is required of the system?

**Specifying the requirements of real-time embedded systems**

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.

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 [4], it may be an extremely onerous task.

To overcome this problem, and to improve software quality and productivity, the use of mathematically based specification languages has been proposed [5,6].

**Formal methods in software engineering**

What are formal methods when applied to software? They are rigorous engineering practices which are generally based on mathematical formal systems and which are to be applied to the development of software [7]. More concisely, they use a mathematically based formal notation to describe the structure, function and behaviour of software systems.

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 [8], for instance, embraces the cycle from specification through to coding. Others, such as FOREST [9], address themselves only to the problem of stating system specifications. Here we are concerned mainly with the specification aspects of formal methods.

There are a number of claimed benefits for the use of formal specification techniques. First, they introduce precision, rigor 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 question then arises concerning who should produce the formal specifications. Few systems designers have knowledge or experience of discrete mathematics, the basis of formal specification languages. 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. How then can the systems designers 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 discharge highlighted. To illustrate this point, appendix A of this paper contains the discharge of the implementability proof for a very simple logic function (the specification being given in fig.1). Its intricacy raises a serious question. How confident can we be in our ability to detect errors in such mathematical workings?

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 [10]. It provides a demonstration of executable specifications ('animates the specification') in terms of the SYSTEM domain.

Input1

Input2

Output

Logic = { High, Low }

And_Gate (Input1:Logic, Input2:Logic) Output:Logic
ext
pre true
post ( Input1 = High and
       Input2 = High and
       Output = High )
  or ( ( Input1 = Low or
        Input2 = Low ) and
       Output = Low )

Figure1. Specification for a logic function

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.

Animation prototyping, when applied to formal specifications, provides a pictorial description of the meaning of these specifications. Hence it removes the comprehension problem faced by system designers when presented with documents written in a formal notation. However, it can only work successfully provided it meets the following criteria:

- The animation MUST correctly represent the formal specification.
- It must be possible to build the animation model quickly.
- The client (system designer) should need only a minimal knowledge of software and formal notation to understand the model behaviour.

Animation prototyping of VDM specifications

Currently there is extensive research work being carried out into formal methods, the main areas of interest for real-time systems being:

- Specification and verification of timing properties.
- Specification of concurrent processes.
- Specification and analysis of safety-critical systems.

The result has been a proliferation of notations. In the UK the most established methods are VDM, Z [11], OBJ [12] and FOREST. VDM is used in this research project for two reasons. First, it is a mature, well documented, technique. Second, it has been used by the industrial collaborator of this project to formally specify systems for more than five years.

The complete VDM specification language is not inherently suitable for animation. Fortunately, the static behaviour of practical real-time embedded systems can be described using only a relatively small subset of VDM. There are though, no facilities for embedding time/event dependencies within the specification language at the present time. This is not a problem in the current application, but is something which must be solved if VDM is to be applied generally to real-time systems.

The system specifications are built in model-oriented style, comprising:

- Definition of the set of system operational states.
- Definition of all possible initial states.
- Collection of operations whose external variables are parts of the states.

The definition of operations is that given by Jones [8]. Only the simple logical operators are used as the systems being specified do not involve complex dynamic data structures.
Rather, the emphasis is on the correctness of the control structure and the ordering of events.

The design approach used here is to specify requirements in a top-down hierarchical manner; operations are decomposed using the standard structured programming constructs of sequence, selection and iteration. Further, in real-time systems the ordering of operations is important. VDM by itself does not define the ordering of specifications; it is enforced here by the design method used with the animation process. A diagramming design technique is used, as shown in fig.2.

Correctness proofs are an essential part of any formal method. It must be possible to mathematically prove that decomposed specifications correctly describe the higher-level ones. The notation and style developed here allow such formal proofs to be constructed (though this is not the main aim of the project). The decomposition can also be demonstrated to be correct informally through the use of animation.

Practical issues in animating formal specifications

Some work has already been done within the field of formal specifications to derive executable prototypes from such specifications [13, 14]. The main thrust of the work described here is that it aims to express such executions in system, not computer, terms. Therefore model building techniques are needed to cope with the wide diversity of real-time systems. And, most importantly, prototypes must be generated automatically from the formal specifications. Otherwise there is no guarantee that the prototype accurately demonstrates the formally specified properties of the system.

The method developed here has been designed to cater for a particular category of systems, having the following attributes:

- Hard real-time performance.
- Safety-critical applications.
- No interrupt functions allowed.
- Executives/schedulers not permitted.
- All operations strictly sequential.

These last three items are a consequence of the safety-critical function.

The basic requirement of the animation tool is to translate VDM specifications into an animated prototype. But it must also provide extensive facilities for building icon-based dynamic models of the system. For these reasons SIMSCRIPT II.5 has been used as the development language. The screen displays are built using standard SIMSCRIPT constructs, whilst animation of these is provide by SIMSCRIPT code ('animation code') derived from the VDM specifications.

The animation code is derived in the following manner (see fig.3). First, the system specification structure chart is produced by a specialist in formal methods. Next a text file is generated containing the related VDM specifications. After this the

![Figure 2 Specification Structure Diagram](image)

![Figure 3 Specification to Animation Translation](image)
operation declarations and operation decompositions are translated into SIMSCRIPT routines, together with their corresponding control structures. Finally these are integrated with the screen display/user interaction software.

Demonstration system

The techniques described above have been applied to the animation of the specifications for a small chemical plant. This is a nitrogen/hydrogen compressor system, involving sequence control, monitoring and alarm functions. Its specifications are formally defined in a structured hierarchical manner using a subset of VDM. Part of these is given in fig.4. Decomposed specifications have been rigorously proven to be correct. Translation of the VDM specification to diagram animation is fully automated. Animation can be performed at any level in the decomposition using these specifications.

Concluding comments

The automated animator has been demonstrated to a wide range of engineers, including project managers, systems designers and design specialists. The reaction to these presentations has been extremely encouraging, reinforcing our belief in the use of animation techniques.

Current work is concentrated on developing a fully functional animator for use in safety-critical real-time systems. The longer term research aims are to extend VDM notation to accommodate time/event dependencies as encountered in embedded real-time systems.

Acknowledgements

Acknowledgements are due to CACI Inc. for the provision of the simulation package SIMSCRIPT II.5 (SIMSCRIPT II.5 is a registered trademark and service mark of CACI Inc.).

References


APPENDIX A: Implementability proof for And operation.

from I1, I2 ∈ Logic × pre-And(I1, I2) ⇒
3 O ∈ Logic  post-And(I1, I2, 0)
1 from X ∈ Logic
1.1 X = High v X = Low Logic-defn(h1)
1.2 ¬(X = High) v ¬(X = Low) Logic-defn(h1)
1.3 ¬(X = High) → X = Low =-defn(1.2)
1.4 X = Low → ¬(X = High) =-defn(1.2)
1.5 (¬(X = High) → X = Low )
∧ ( X = Low v ¬(X = High )) =-I(1.3, 1.4)
infer ¬( X = High ) ⇔ X = Low =-I(1.5)
2 from I1, I2 ∈ Logic
2.1 from I1 = High ∧ I2 = High
2.1.1 ¬( I1 = High ∧ I2 = High ) =-I(h2.1)
2.1.2 ¬( ¬(I1 = High) v
¬(I2 = High) ) de-M(2.1.1)
2.1.3 ¬( I1 = Low v I2 = Low ) =t-sub(2.1.2.1)
2.1.4.1 ¬( I1 = Low v I2 = Low v)
∧ ( I1 = High ∧ I2 = High ) =-I(2.1.3, h2.1)
infer I1 = Low v I2 = Low =-I(2.1.4)
2.2 from I1 = High ∧ I2 = High
2.2.1 ( I1 = High ∧ I2 = High ) v
¬( I1 = High ∧ I2 = High ) =-I(h2.2)
infer I1 = High ∧ I2 = High v
( I1 = Low v I2 = Low ) =t-sub(2.1)
2.3 from O ∈ High
2.3.1 from O ∈ High
2.3.1.1 (I1=High ∧ I2=High) ∧ O=High =-I(h2.3, h2.3.1)
2.3.1.2 (I1=High ∧ I2=High) ∧ O=High
∧ (I2=Low ∧ I2=Low) ∧ O=Low =-I(h2.3.1.1)
infer post-And(I1, I2, 0) =s-sub(2.3.1.2)
2.3.2 High ∈ Logic
infer 3 O ∈ Logic post-And(I1, I2, 0) =-I(2.3.1, 2.3.2)
2.4 from O ∈ Logic post-And(I1, I2, 0)
2.4.1 from O ∈ Low
2.4.1.1 (I1=Low v I2=Low) ∧ O=Low =-I(h2.4, h2.4.1)
2.4.1.2 (I1=Low ∧ I2=Low) ∧ O=Low
∧ (I1=Low v I2=Low) ∧ O=Low =-I(2.4.1.1)
infer post-And(I1, I2, 0) =s-sub(2.4.1.2)
2.4.2 Low ∈ Logic
infer 3 O ∈ Logic post-And(I1, I2, 0) =-I(2.4.1.2)
2.5 from O ∈ Logic post-And(I1, I2, 0)
2.5.1 from O ∈ Logic post-And(I1, I2, 0)
2.5.2 infer pre-And(I1, I2) ⇒
3 O ∈ Logic post-And(I1, I2, 0) =-I(2.5)
infer ∨ I1, I2 ∈ Logic pre-And(I1, I2) ⇒
3 O ∈ Logic post-And(I1, I2, 0) =-I(2)