



Documenting the software lifecycle using formal methods

D.M. Ayres

*AEA Technology, Winfrith Technology Centre, Dorchester,
Dorset DT2 8DH, UK*

ABSTRACT

Much has been claimed about the benefits of using formal methods to obtain reliable software. However, software quality is not only concerned with writing software which is correct, but with writing software which is *demonstrably* correct. This paper shows how formal methods can be used as a tool for developing software and for providing clear step-by-step documentation of the development processes.

The role of formal specification in requirements capture is discussed, with emphasis on the use of a formal notation as a tool for thinking about, communicating and recording specifications as they evolve from a nebulous initial concept to a specification which may be developed into code. Consideration is given to techniques for validating a specification; for demonstrating that relevant properties of the system are captured in the specification.

The benefits of applying formal development techniques is examined. Proof obligations are introduced as a way of documenting design decisions and providing a clear record of the development process.

A small example of a Pascal procedure developed using the Vienna Development Method (VDM) is used to illustrate the techniques described above. Relevant tool support is examined.

The costs of applying formal techniques are examined, together with an appraisal of the factors which have prevented the widespread industrial use of formal methods to date.

INTRODUCTION

Software failures have placed a large financial cost on world industry, and with more and more software being used in safety-critical applications, the

648 Software Quality Management

impact of failure could pose a threat both to human life and to the environment. Clearly, every effort must be made both to ensure that software is correct, and to *demonstrate* to the outside world that it is correct. The latter is as important as the former, for safety-critical systems have to undergo an independent assessment before being granted a licence to operate by the appropriate statutory body.

When we say that we must demonstrate that software is correct, we find a difficulty; the term 'correct' can be rather hard to define. Presumably, by correct we mean 'complies with its specification'. However, most specifications that are written in a natural language such as English suffer from two difficulties:

- Natural language specifications are difficult to organize in such a way that each facet of the system is described in one place only. Multiple references or allusions to the same behaviour can be confusing, and can mask inconsistencies.
- Natural languages are inherently ambiguous. Whilst humourists make a virtue of deliberately exploiting ambiguity, it is a hazard which specification writers must avoid. However, avoiding ambiguity is not easy, and is compounded by the problem of having multiple references to the same behaviour.

In an attempt to clarify ambiguity in natural language specifications, examples are often used. Unfortunately, these can prejudice the developer towards particular design decisions, thus compromising the specification.

Structured techniques of software engineering such as the Yourdon technique provide the means of breaking the specification into entities; conceptual units that are more easily thought and reasoned about than the monolithic specification. They also provide methods for designing each conceptual unit and producing code.

Structured techniques have much to commend them; they provide traceability and are easy to learn. They make it possible to manage large projects which are worked upon by teams of people. However, they suffer from drawbacks. The highest level presentation of the requirements is divided into entities which can impose a structure on the resulting code. This division can not be termed a specification, as decisions about high-level structure have already been made. Thus a higher level, more abstract specification would be desirable. Such a specification would contain less design detail, and so be more easily understandable.

Relationships between different levels of the design, and between the specification and design are often presented by graphical means or by struc-

tured design documentation which gives an intuitive feel to the design approach. Ultimately, the design decisions which have been made in each level have to be discovered by comparing the design documents and by reading the related text. Thus the documentation of each step in the design may be poor. There is no concise notation for documenting design decisions and expressing the relationships between each level of design.

The structured method SSADM provides a system model around which iterative design can take place. Checking operations are employed between design steps to help verify successive stages, but this is a purely manual process. SSADM is described in SSADM [1].

THE VIENNA DEVELOPMENT METHOD—VDM

Formal methods for specification and design have at their heart a notation with a mathematically defined syntax and semantics. This is usually augmented by some logical framework within which specifications and designs can be reasoned about, and by a rationale for design. Each step in the design can be verified by proving its correctness.

The Vienna Development Method is one of the most widely used formal methods available at the current time. It has been used by AEA Technology on several research projects. The remainder of this paper is concerned particularly with VDM, though much of the principles discussed are equally applicable to other formal methods.

So what do formal methods, in particular VDM, have to offer over the structured methods? Firstly, the mathematical nature of the notation means that it is not inherently ambiguous in the way that natural languages are. Each facet of behaviour can be described using one or more cooperating mathematical functions, which are referenced by other parts of the specification as necessary. Thus descriptions are localised rather than being spread throughout the document, making it easier to avoid the problem of multiple descriptions of the same behaviour. The precision of formal notations make them ideal for use when a team of people are eliciting requirements. They can be used to communicate ideas without danger of misunderstanding.

Abstraction can be used to avoid introducing into the specification behaviour or properties which are irrelevant from the system requirements perspective, and to deliberately under-define behaviour in cases where the requirements do not completely determine behaviour.

Formal specifications can be subject to various validation techniques. Possibly the simplest is to animate the specification. This may be achieved by use of an executable formal specification language or by prototyping to create an executable which demonstrates the behaviour of the specification. This technique is particularly useful for demonstrating the specification to

650 Software Quality Management

the customer or the requirements author. Another validation technique is to posit desirable properties in the formal notation and deduce that they hold by use of formal proof.

Possibly the most enticing prospect of using formal methods on safety-critical software is the potential to formally prove that the software meets its specification. However, formal proof can be time-consuming and costly. Theorem proving assistants and proof checkers are essential for tackling industrial-scale proofs, but there are few mature tools around. Thus, at the present time, full proof of all but the smallest subsystems is something which must be approached with caution.

When verifying a design step, one or more *proof obligations* to support that step are constructed. A proof obligation is a set of hypotheses and a conclusion, all of which are written in a logical notation. A proof obligation is said to be *discharged* when the conclusion is deduced from the hypotheses by the application of a sequence of logical inference rules. The proof of the obligation is the sequence of deductions by which the conclusion is derived. When all the proof obligations associated with a design step are proved, then the design step is verified.

Proof Obligations as Design Documentation

Despite the difficulties associated with discharging proofs, there is much benefit to be gained from generating proof obligations. Proof obligations can be considered as statements which document the design step. They record the design decisions which have been made in the step. If there is any doubt about the correctness of a step, then a full formal proof can be constructed.

The use of proof obligations as design documentation also increases maintainability after the system is commissioned. They provide a clear, unambiguous statement of the design decisions made during the creation of the software.

A SMALL EXAMPLE SYSTEM

In order to illustrate the notion of proof obligations as documentation for the development process, a small example is instructive. We will take a small part of the specification of a nuclear reactor protection system. This protection system, though not used on any real plant, is a realistic system. It was used as an exercise by AEA Technology to examine the feasibility of formal specification using VDM.

For our example here, we will consider this fragment from the natural language specification of the full system:

The system should accept a set of integers as input and re-

turn the largest integer in the set.

This is a simple specification, and one that seems to be well-defined, complete and unambiguous. However, we will show that in its present form, it is impossible to implement on any machine.

We will use it as the basis of an example to illustrate the use of VDM for clarifying requirements, designing and coding in Pascal.

Specification in VDM

We can write the requirement above as a VDM function:

```
max1( $m : \mathbb{Z}\text{-set}$ )  $r : \mathbb{Z}$   
pre  $m \neq \{\}$   
post  $r \in m \wedge \forall s \in m \cdot s \leq r$ 
```

The first line of the VDM defines a function named *max1* which takes as input a set of integers *m* and returns an integer *r*. The line beginning with the keyword *pre* is a *precondition*. The function can only be expected to work if the precondition is true (in this case, if *m* is not the empty set). The *postcondition* is on the line that begins with the keyword *post*, and is the condition which holds after the function has completed. In this case it states that the integer *r* must be in the set *m*, and that every integer in *m* is less than or equal to *r*. These two conditions together are sufficient to ensure that the function *max1* returns the largest integer in the input set.

Thus we have specified the problem in a general way, without specifying the manner in which the maximum value will be calculated, and without specifying the way in which the abstract notion of a set will be implemented on a machine. Such detail has no place in a specification, and should be confined to the design stages.

A Problem with the Requirements Our function *max1* takes as its argument any member of $\mathbb{Z}\text{-set}$, the set of finite subsets of integers. This poses a problem: we wish to implement our specification on a machine which has a finite storage resource available. No matter how large that resource may be, our specification requires that we be able to store sets of numbers which exceed the available storage resources. This indicates that our specification is inadequate, as it is impossible to implement correctly.

In reality, our function is never required to accept sets containing more than 28 numbers. This information was deduced by examining the specification of the reactor protection system to identify the manner in which our function is used. This illustrates the problem with natural language

652 Software Quality Management

specifications described earlier; namely that information relating to an aspect of behaviour can be spread throughout a large specification. It is quite common for the requirements to be refined as a result of formal specification work.

It is clear that we must re-examine our specification and modify it to work on sets of bounded size. We define a constant value ms which is defined to be the maximum cardinality of sets upon which our function must operate. We define a type *Class* of allowable sets of integers. We use an invariant to restrict *Class* to contain sets of cardinality no greater than ms .

$Class = \mathbb{Z}\text{-set}$

$\text{inv } c \triangle \text{card } c \leq ms$

$ms : \mathbb{N} = 28$

$\text{max2}(m : Class) r : \mathbb{Z}$

$\text{pre } m \neq \{\}$

$\text{post } r \in m \wedge \forall s \in m \cdot s \leq r$

Documenting the Relationship between *max1* and *max2* We can write proof obligations which describe the relationship between the two specifications.

Note that we have done no more than restrict the domain of the function. i.e. we have reduced the domain of *max2* to *Class*, which is a subset of $\mathbb{Z}\text{-set}$, the domain of *max1*. Thus we expect two conditions to hold, called Domain and Result rules:

max1-max2 Domain We expect to be able to apply *max1* to every set upon which we may apply *max2*

max1-max2 Result Provided we can apply *max2* to a set, then the result of applying *max1* to that set is the same as the result of applying *max2*.

This is not the most usual form for presenting proof obligations, as it is imprecise. Instead we write them as follows:

$$\boxed{\text{max1-max2 Domain}} \quad \frac{m : \mathbb{Z}\text{-set}}{\text{pre-max2}(m) \wedge \text{inv-Class}(m) \Rightarrow \text{pre-max1}(m)}$$

$$\boxed{\text{max1-max2 Result}} \quad \frac{m : Class, \quad r : \mathbb{Z}}{\text{pre-max2}(m) \wedge \text{post-max1}(m, r) \Rightarrow \text{post-max2}(m, r)}$$

Above the line there is a list of predicates separated by commas. These are the hypotheses. Below the line is the conclusion, a predicate which must be deduced from the hypotheses to discharge the proof obligation.

The proof obligations were constructed by examining the change in the domain (the input space) of the function, and by examining the change made to the result of the function. Each consideration gives rise to one proof obligation. Although these proof obligations were constructed by eye, they are of a form which is very common.

This pair of proof obligations embodies the nature of the change we have made to the requirements. They provide a precise statement of the relationship that exists between the two specifications. The obligations may be proved formally, or justified by informal argument. All the proof obligations presented in this document have been discharged formally using the mural proof assistant. An informal justification of the **max1-max2 Result** obligation may proceed like this:

From the hypotheses, c is of type *Class* and r is of type $\mathbb{Z}$. Thus, by definition of *Class*, c is a set of integers which satisfies the condition $\text{card } c \leq ms$.

Let us suppose that $\text{pre-max2}(c)$ and $\text{post-max1}(c, r)$ both hold. Then, by definition of post-max1 ,

$$\begin{aligned}\text{post-max1}(c, r) &\Rightarrow r \in c \wedge \forall s \in c \cdot s \leq r \\ &\Rightarrow \text{post-max2}(c, r)\end{aligned}$$

□

If at a later date the validity of this proof obligation were questioned, then the outline argument given above could be replaced by a fully formal proof.

Design of the **max2** function

We have illustrated how VDM can be used to record changes in a specification as a result of requirements changes. We now present a design decision, and derive proof obligations to document it.

The VDM specification *max2* uses a set data type. Few high-level languages support set types directly, and those such as Pascal which do support sets only provide basic operations on them, and the size of the sets is strictly limited. Thus we will make a decision to implement set types as arrays. This process is known as *Data Reification*. ‘Reify’ means literally ‘to make concrete’. Data Reification is described in detail in Jones [2].

Our chosen implementation language is Pascal. Unfortunately, Pascal doesn’t support variant arrays. Thus we must use arrays of fixed length *ms*.

654 Software Quality Management

To support sets which are smaller than the maximum size, we also require a variable *bound* which indicates the size of our set. We will use the first *bound* elements of the array to store a set of *bound* elements.

VDM contains a sequence type, which is very similar to a Pascal array. We use Int^* to denote the set of finite sequences of integers. We will use a VDM composite type to link the array with *bound*:

$$\begin{array}{ll} \text{Array} :: & ar : Int^* \\ & bound : Nat \\ \text{inv}(m) \triangle & m.bound \leq ms \wedge \text{len } m.ar = ms \end{array}$$

Note that if m is of type *Array*, then we access its components by $m.ar$ and $m.bound$.

The representation we will use is simple: Suppose ms takes the value 5.

{2}	is represented by	[2, 0, 0, 0, 0],	$bound = 1$
{89, 7, 14}	is represented by	[89, 7, 14, 0, 0],	$bound = 3$
{1, 3, 5, 7, 9}	is represented by	[1, 3, 5, 7, 9],	$bound = 5$

Note however that

{34, 17}	is represented by	[34, 17, 0, 0, 0],	$bound = 2$
	and also by	[17, 34, 0, 0, 0],	$bound = 2$

Also,

{4, 12, 34}	is represented by	[4, 12, 34, 0, 0],	$bound = 3$
	and also by	[4, 12, 34, 0, 1342],	$bound = 3$
	and even by	[12, 4, 32, 17, 397, -65],	$bound = 3$

To convert from the array representation to the set representation we simply take the first *bound* elements of the array and place them in a set. This is formalized by defining a *retrieve function*. The retrieve function takes a member of the type *Array* as argument and retrieves the corresponding member of the type *Class* which it represents.

$$\begin{array}{l} \text{retr-Array} : \text{Array} \rightarrow \text{Class} \\ \text{retr-Array}(m) \triangle \text{elems}(m.ar)(1, \dots, m.bound) \end{array}$$

We now have a precise definition of our reified data type *Array* and its relationship with the original type *Class*. Before we can proceed, we must verify this design step. We must verify the *adequacy* of the reification. That is, given any member of the type *Class*, can we find a member of the type *Array* which represents it. This yields the **Array-Adequacy** proof obligation:

$$\boxed{\text{Array-Adequacy}} \frac{c: \text{Class}}{\exists m: \text{Array} \cdot \text{retr-Array}(m) = c}$$

The proof of this obligation is rather tricky; it is best proved by induction on the cardinality of the set c . However, we need to create a theory of sequences in which no elements are repeated. Nevertheless, the obligation can be discharged formally, or an informal argument applied.

The data reification has consisted of making a design decision to represent sets by arrays. The adequacy obligation and the retrieve function together document this design decision precisely.

Recasting *max2* in terms of the type *Array* Having reified the data type, we can now rewrite *max2* in terms of the new type. This is not difficult. We simply replace occurrences of set notation with array notation acting upon the corresponding components of the *Array*. Although this rewriting may seem a little ad-hoc, it is really a simple translation. The bulk of the work was done in creating the retrieve function and verifying the adequacy obligation.

$$\begin{aligned} & \text{max3}(m : \text{Array}) \ r: \mathbb{Z} \\ & \text{pre } m.\text{bound} > 0 \wedge m.\text{bound} \leq ms \\ & \text{post } r \in \text{elems } m.\text{ar}(1, \dots, m.\text{bound}) \wedge \forall s \in m.\text{ar}(1, \dots, m.\text{bound}) \cdot s \leq r \end{aligned}$$

We can generate proof obligations to verify our translation. There are two proof obligations; a domain obligation and a result obligation. The form of these obligations is defined and justified in Jones [2]. To summarize, the domain obligation shows that *max2* and *max3* can be applied to equivalent classes of data. The result obligation shows that given any member of the type *Array*, the answer we get by applying *max3* and retrieving it is the same answer that we get by first retrieving and then applying *max2*.

Here are the obligations:

$$\boxed{\text{Array-Domain}} \frac{m: \text{Array}}{\text{pre-max2}(\text{retr-Array}(m)) \Rightarrow \text{pre-max3}(m)}$$

$$\boxed{\text{Array-Result}} \frac{(m: \text{Array}), (r: \mathbb{N})}{\begin{aligned} & \text{pre-max2}(\text{retr-Array}(m)) \wedge \text{post-max3}(m, r) \\ & \Rightarrow \text{post-max2}(\text{retr-Array}(m), r) \end{aligned}}$$

Coding in Pascal

We can implement our specification very simply. A loop is used to examine each element of the array, selecting the biggest element seen so far. The code is shown in figure 1.

656 Software Quality Management

```
CONST
  ms = 28 ;

TYPE
  Scope = 1..ms ;
  Seq = array[Scope] of Integer ;
  Array = RECORD
    ar    : Seq ;
    bound : Integer
  END ;

FUNCTION max (c: Array): Integer ;
  VAR i  : Integer ;
      ma : Integer ;

  {pre c.bound > 0}
  BEGIN
    IF c.bound = 1 THEN ma := c.ar[1] ELSE
    BEGIN
      i := 1 ;
      ma := c.ar[i] ;
      REPEAT
        {assert ma >= c.ar(i)
          and i >= 1 and i < c.bound
          and for_some(q:Scope, 1 <= q and q <= i
                                and c.ar[q] = ma)
          and for_all(r:Scope, 1 <= r and r <= i
                                -> c.ar[r] <= ma)    }
        i := i + 1 ;
        IF c.ar[i] > ma THEN ma := c.ar[i] ;
      UNTIL i = c.bound
    END ;
    max := ma
  END ;
  {post for_some(k:Scope, 1 <= k
                    and k <= c.bound and  max = c.ar[k])
    and for_all(r:Scope, 1 <= l
                    and l <= c.bound -> c.ar[l] <= max    }
```

Figure 1: Pascal Implementation of *max3*

We start by setting `ma` to the first element of the array. Then we examine each element in turn, setting `ma` to the value of the current value if it is bigger than `ma`. After the loop has terminated (which must happen since on each execution we increase the value of the loop variable `i`, starting at 1 until we reach `ar.bound`), `ma` contains the maximum value.

Notice that the executable code has been supplemented by comments containing pre- and postconditions, and an assertion in the loop. These comments are in a format similar to that which can be used by the SPADE toolset. Among the SPADE tools is a Verification Condition Generator. This generator will generate a series of verification conditions (proof obligations) which relate the action of the code to the formal specification. These can be proved using another SPADE tool, or may be used as documentation of the implementation stage.

CONCLUSIONS

The example above has shown that every stage of development of a system from requirements analysis to implementation in a high level language can be conducted using VDM. We have demonstrated three main benefits to this approach:

1. Formal specification languages are a powerful tool for capturing requirements. The mathematical precision of the notation together with the potential for abstraction allows specifications to be written in a clean, unambiguous style which does not contain any implementation-level clutter.

The existence of a precise, validated formal specification at an early stage in the lifecycle makes it relatively easy to construct a software test plan.

2. Design decisions are documented precisely and clearly by proof obligations. If the correctness of a design step is in doubt, then an attempt can be made to discharge the relevant proof obligations. If they are proved, then the correctness of the design step is also proved.

Maintenance of the product is improved by clear documentation of each of the design steps made in the development.

3. Initially, proof obligations can be accompanied by informal arguments to give confidence that a design step is correct. However, these informal arguments can be replaced by formal proofs throughout in order to prove that the code correctly implements its specification.

658 Software Quality Management

Drawbacks of Formal Methods

The clear benefits of using VDM described above beg an obvious question: *why are formal methods not widely used in industry?*

There are many reasons for this, some of which are serious considerations, others of which are unfounded.

Skills Required To apply a formal method requires a knowledge of some basic mathematics such as set theory and predicate calculus. Although many people become very worried at the sight of a 'backwards-E' or an 'upside-down-A', this fear is usually overcome because the level of knowledge of these mathematical disciplines required by someone who wishes to use VDM is not great. Any numerate software designer should be able to learn the necessary skills with ease.

The 'culture shock' suffered by experienced software engineers is perhaps of more importance. The use of VDM for specification requires a shift in thinking away from a programmers' expectation that a specification says how a task is to be performed and in what order. An implicit VDM specification such as the example shown here provides information about the state of the system before an operation is performed, and information about the state of the system after the operation has terminated, but says nothing about how it moves between the two states. Experience in-house suggests that when traditional software engineers are presented with a formal specification for the first time, they attempt to read it like a program, bringing with them a baggage of preconceptions with which they try to interpret more information from the specification than it is intended to convey. However, with practice and perseverance, this can be overcome.

Cost There is a great deal of evidence which suggests that formal development is more expensive than structured approach, and almost as much evidence to the contrary. Most of this evidence is anecdotal, and so it is difficult to come to a considered opinion.

It is clear that the construction of formal proof is both time-consuming and requires specialist skills. However, there are many benefits to be gained from employing formal methods even if few or no proof obligations are formally discharged. Experience within AEA Technology suggests that without proof, formal development may be a little more expensive than structured development, but the cost is easier to predict. This is a result of most requirements errors being detected during the formal specification phase. If these errors are not detected early, they can be the source of expensive rework during later phases. This cannot be predicted in advance.

The DARTS project is an ESPRIT II project which is being undertaken by a consortium lead by AEA Technology. Four implementations

of an example reactor protection system have been designed and implemented, each using a different methodology, of which two are based upon formal methods and two upon structured methods. Data about effort expended and errors introduced was collected during development. The four implementations are being executed in parallel to highlight erroneous behaviour. When the results of the experiment are published, some justified comparisons of the cost of formal methods will be available. Smith+ [3] contains a detailed description of the DARTS experiment.

Tool Support The manipulation of long strings of mathematical symbols, whether in a specification or a proof, is inherently prone to human error. Thus the provision of tool support is an essential requirement for the adoption of formal methods.

Three tools have been used in the preparation of the worked example.

- SpecBox is an interactive, window-based tool which supports the preparation of specifications in the draft BSI/ISO standard definition of VDM. The tool is a commercial product which is developed and marketed by Adelard. It provides syntactic checking of specifications in the ASCII syntax. Various semantic checks are also provided. SpecBox contains a $\text{\LaTeX}$ generator which produces a source file containing the specification which has been translated into the mathematical syntax. This feature was used directly in the preparation of this document. The system also includes an interface to the mural proof checker. More information about SpecBox can be found in Froome+ [4].
- mural is a system which supports reasoning about specifications. Its main component is a proof assistant. This consists of a library of axioms and derived rules together with a proof tool (illustrated in figure 2) which provides facilities for entering and discharging proof obligations. A detailed technical description of mural is available in Jones+ [5].
- The final tool used for the preparation of the example is the SPADE Verification Generator and Proof Checker. This provides an environment for checking that implementations in subsets of languages such as Pascal meet their formal specification.

The majority of tools available today were developed within Universities as items of academic interest. Consequently, many are restricted to subsets of notations, cannot handle large problems, and are very slow to use.

Academics generally have little interest in the commercial exploitation and development of tools, and the potential user base is currently so

Justification Tool					
justif type	rule	seqHyp	foldDef	unfoldDef	oracle
hyps	3				sel
concl	4				
select or search		select			
select rule / instantiate					
rules	insts	meMap : { cr $\rightarrow$ r }			
retr-Classrep defin	1	mtMap : {}			
		{ cr $\rightarrow$ cr }			
		{ [cr] { (cr : Classrep) } $\vdash$ ((elems [(s-ar [cr])] : Class)) { (r : Classrep) }			
		{ ((retr-Classrep [r]) = (elems [(s-ar [r])])) }			
instantiate					
new lines					
(r : Classrep)					
new conclusion					
top level		add to proof			

Figure 2: The mural Proof Assistant in Use.

small that commercial companies are unwilling to invest heavily in the development of industrial-scale tools. These factors together mean that the development of industrial tools is a slow process. However, SpecBox has shown that tool support can be made commercially viable, and this could prepare the way for others.

Scaleability The final factor which has affected the slow take-up of formal methods within industry is the fear that formal methods are unworkable on large problems.

The example described above is a tiny part of a full system, and yet its development has generated 5 proof obligations, together with 8 code verification conditions. We have estimated that the complete example reactor protection system from which this example was drawn would give rise to several hundred obligations. Empirical arguments suggest that the number of proof obligations increases approximately linearly with the number of functions and operations which make up the specification.

As projects become so large that teams of people are working on them, there is a need for multi-user tools and environments. These are not currently available.

References

- [1] SSADM Version 4 Reference Manual, CCTA, UK, 1990
- [2] Jones, C.B. *Systematic Software Development using VDM*, Prentice-Hall, London, 1990
- [3] Smith, I.C., Wall, D.N. and Baldwin, J.A. 'DARTS—An experiment into cost of and diversity in safety critical systems,' in SAFECOMP'91 (Ed. Lindberg, J.F.), pp35–40, *Proceedings of the IFAC/IFIP/EWICS/SRE Symposium*, Trondheim Norway, 1991. Pergamon Press.
- [4] Froome, P.K.D., Monahan, B.Q. and Bloomfield, R.E. 'SpecBox—a checker for VDM specifications,' in *Proceedings of Second International Conference on Software Engineering for Real Time Systems*, Cirencester, IEE 1989.
- [5] Jones, C.B., Jones, K.D., Lindsay, P.A. and Moore, R. *mural: A Formal Development Support System*, Springer-Verlag, New York, 1991