FORMAL METHODS OF SOFTWARE DEVELOPMENT ADVANCES AND RETREATS

D. C. Ince

Abstract: This paper is about the use of discrete mathematics within software development. It describes, in outline, how discrete mathematics can be used to specify large computer systems, and how mathematical proof can be used to validate a system. This area of computer science is exceptionally promising, but is prevented by major problems from being adopted on industrial software projects. The paper examines one problem: that of data refinement and outlines one possible solution. It concludes by briefly examining the advances that have been made in formal methods of software development and also looking at where progress has been slow.

1. Introduction

Modern software development projects are normally organized on a phase-by-phase basis. One popular model is shown in Figure 1. Here the development process is split up into a number of separate activities, with each activity delivering a document which then forms the input into the next activity. The activities shown are:

Requirements analysis. This is the process of eliciting the requirements of a system from a customer. The requirements will be a mixture of functions: descriptions of what a system is intended to do, and constraints: statements which

This article is the text of an invited lecture given by the author at the September meeting of the Irish Mathematical Society held at the Regional Technical College, Waterford, September 3–4, 1992.

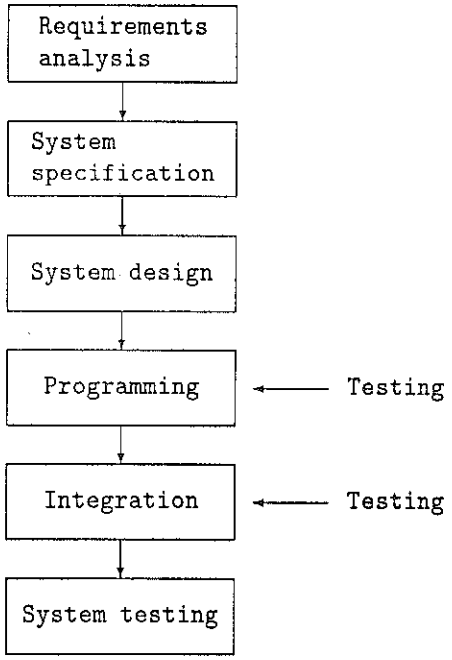

Figure 1: Conventional software development

constrain the system to be produced, or the process of developing the system. An example of the former is a constraint that a certain response time is required; while an example of the latter is the fact that the developer should use some particular programming language.

System specification. This is the process whereby the properties of the system which were discovered during the process of requirements analysis are written down. The document which is produced by this task is known as the *system specification*, although sometimes it is referred to as the *requirements specification*. Normally it is expressed in natural language.

System design. This is the activity in which the system specific-

ation is used to guide the process of deriving an overall system architecture. The architecture being expressed in terms of modules (subroutines, procedures, programs).

Programming. This is the process of taking the individual modules defined during system design and expressing them in some form of programming language.

Integration. The process of bringing together the programmed modules to form a final system.

Accompanying these development tasks is a set of parallel tasks which have the aim of validating the system: checking that user requirements are encapsulated in the system, and that individual software tasks such as integration have been carried out carefully. Examples of such tasks are: system testing, acceptance testing and module testing. This is the model of development that has been used for over twenty five years. However, the documents that are used for software development and which are generated by developers can be very flawed, leading to budget overruns, time overruns and even the cancellation of projects. In order to look at the problems which occur with these documents it is worth looking at the system specification.

2. Problems with the system specification

Although I am using the system specification as an example it is worth stressing at this juncture that similar problems occur with all the document produced by a software project. Life would be uncomplicated for the software developer if the system specification consisted of a series of sections marked

- Functional requirements;
- Non-functional requirements;
- Goals;
- Data requirements;
- Implementation and design directives;

each of which were consistent, unambiguous, and complete and where the text would be expressed in user terms. Unfortunately, this very rarely happens. The purpose of this section is to outline how reality deviates from the ideal.

In general, a system specification will be vague, contradictory, incomplete, and will contain functional requirements, constraints, and goals randomly mixed at different levels of abstraction. Often, it will either have a very naïve and over-ambitious view of the capabilities of a software system or a view which was current a few decades ago.

Vagueness

A system specification can be a very bulky document and to achieve a high level of precision consistently is an almost impossible task. At worst it leads to statements such as

> The interface to the system used by radar operators should be user friendly.
>
> The virtual interface shall be based on simple overall concepts which are straightforward to understand and use and which are few in number.

The former is at too high a level of abstraction and needs to be expanded to define requirements for help facilities, short versions of commands, and the text of user prompts. The latter is a platitude and should be removed from the specification.

Contradiction

A system specification will often contain functional and non-functional requirements which are at variance with each other. In effect they eliminate the solution space of possible systems. Typically, the sentences that make up the contradictions will be scattered throughout the document. An extreme example of such a contradiction is the statement

> The water levels for the past three months should be stored on magnetic tape.

(which may form part of the hardware requirements of a future system) and the statement

> The command PRINT-LEVEL prints out the average water levels for a specified day during the past three

> months. The response of the system should be no longer than three seconds.

Obviously, if a slow-storage medium such as magnetic tape is used then the response time will hardly be in the range of a few seconds.

A more subtle error occurs with the statements

> Data is deposited into the employee file by means of the WRITE command. This command takes as parameters: the name of the employee, the employee's department, and salary.
>
> The ENTRY-CHECK command will print on the remote printer the name of each employee together with the date on which the employee's details were entered in the employee file.

which are functional requirements together with the non-functional requirement

> The hardware on which the system will be implemented consists of: an IBM PC with 512k store, asynchronous I/O ports, keyboard, monitor, and 20 Mb hard disc.

Here the assumption made is that the employee file will contain an entry date for each employee. Unfortunately, the WRITE command does not take an entry date as a parameter, and the hardware specified does not include a description of a calendar/clock.

A system cannot be developed which satisfies contradictory requirements. If this were regarded as a pure example of a contradiction, then the ENTRY-CHECK command should be deleted. However, the contradiction could have arisen from a set of incomplete requirements. In this case the WRITE command should be amended to take the entry date as a parameter or the hardware requirement expanded to include a calendar/clock.

Incompleteness

One of the most common faults in a system specification is incompleteness. An example of this follows. It shows part of the functional requirements of a system to monitor chemical reactor temperatures.

> The system should maintain the hourly temperatures from sensors which are attached to functioning reactors. These values should be stored for the past three months.
>
> The function of the AVERAGE command is to display on a VDU the daily temperature of a reactor for a specified day.

These statements look correct. However, what happens if a user types in the AVERAGE command with a valid reactor name but for the current day? Should the system treat this as an error? Should it calculate the average temperature for the hours *up to* the hour during which the command was entered. Alternatively, should there be an hour threshold below which the command is treated as an error and, above which, the average temperature for the current day is displayed?

Mixed requirements

Very rarely will you find functional requirements partitioned neatly into functional requirements, non-functional requirements, and data requirements. Often statements about a system's function are intermixed with statements about data that is to be processed.

Naïveté

Another common failing of a system specification is that it will contain naïve views of what a computer system can achieve. This will be manifested in two ways. First, the statement of requirements will contain directives and statements which underestimate the power of the computer. The most frequent transgressors seem to be electronic engineers with little experience of software who insist on hardware requirements which could be easily satisfied by software at a much lower cost.

Another example of customer naïveté occurs in system specifications for systems which can never be built within budget. Such systems are normally specified because of the low technical expertise of the customer. The most common example of requirements for an impossible system is the specification of a particular hardware configuration and a set of functions which will never meet its performance requirements.

Another example of naïveté occurs when a customer suffers from a grossly ambitious view of what a system is capable of. One consequence of the recent rise in artificial intelligence has been a rash of system specifications which make the predictions of the wilder members of the artificial intelligence community seem almost sage-like.

Ambiguity

Specifications written in natural language will almost always contain ambiguities. Natural language is an ideal medium for novels and poetry; indeed, its success depends on the large number of meanings that can be ascribed to a phrase or a sentence. However, it is a very poor medium for specifying a computer system with precision. Some examples of imprecision are

> The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security VDU and deposited in the login file when an operator logs into the system.
>
> When an error on a reactor overload is detected the *error1* screen should be displayed on the master console and the *error2* screen should be displayed on the link console with the header line continuously blinking.

In the first statement does the word 'it' refer to the password or the operator identity? In the second statement should both consoles display a blinking header line or should it only be displayed on the link console?

Mixtures of levels of abstraction

A system specification will contain statements which are at different levels of detail. For example, the requirement

> The system should produce reports to management on the movement of all goods to and from all warehouses.

and the requirement

> The system should enable a manager to display, on a VDU, the cash value of all goods delivered from a specific warehouse on a particular day. The goods should be summarized into the categories described in section 2.6 of this document.

are at different levels of abstraction. The second requirement forms part of the first requirement. In a well-written statement of requirements the document should be organized into a hierarchy of paragraphs, subparagraphs, subsubparagraphs, etc. Each level of paragraph represents a refinement of the requirements embodied in the next higher level of paragraph. In a poorly written statement of requirements connected requirements will be spread randomly throughout the document.

3. Mathematics and the software project

The problems outlined above have prompted the software engineering community to look for better notations and methods for the main phases of the software project. The research that has been carried out has had two flavours. The first has involved the invention of graphical notations and software tools for such notations — tools known as analyst or designer workbenches. The second thrust has been in the area of developing mathematical notations. Good introductions to these notations can be found in [2] and [9].

The development of formal methods can be essentially seen as a reaction to the vagaries of natural language, and many of the proponents of such methods will cite the fact that the semantics of mathematics is exact. However, there is much more. My claim that formal methods has a part to play within software development is based on its modelling properties. System specifications

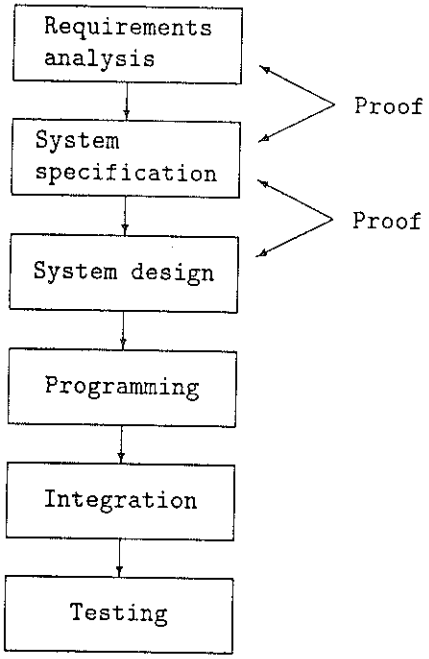

Figure 2: Formal software development

are notoriously cluttered documents, and mathematics enables all the clutter associated with the task of system specification to be removed. The way in which we use formal methods in a software project is shown in Figure 2. It closely mirrors the model put forward in Section 1 of this paper. Requirements analysis is an informal process so it is still carried out in the same way. The difference comes with system specification and design, where a mathematical notation is used to describe a system. Programming remains the same as before. Where the biggest difference is seen is in validation, where mathematical proof is used to check that the system design is a correct reflection of the system specification, and that the program code is a correct reflection of

the system design. Also, mathematical proof is used to explore the consistency of the system specification; for example, in the specification of an editor the analyst would demonstrate mathematically that when an insert command is followed by a delete command which removes the text added by the insert command, the document that is being edited returns to its original state.

It is worth pointing out that system and acceptance testing are still carried out within projects that use formal methods, however our limited experience with formal methods seems to suggest that the amount of reworking that occurs because of a failure of a system or acceptance test is drastically reduced, and the number of system and acceptance tests that fail is also reduced.

Before looking in a little detail at an example of a formal method it is worth stating some of the current problems:

- The size of the proofs that have to be carried out are very large. The mathematics that is produced is quite shallow, but there is quite a lot of it. For an example of the volume of mathematics that is generated see [1].
- There are few tools in existence that effectively support the formal development process. This is a serious problem given the amount of mathematics that has to be carried out.
- The customer has major problems understanding a formal specification.
- The mathematical abilities of many software development staff is not sufficiently sophisticated to use formal methods. To use discrete mathematics as a specification medium requires a high degree of facility in proof, and also the possession of modelling skills which many analysts, designers and programmers do not possess. I would regard this problem as the most serious, and the reason why, I suspect, formal methods will have limited use on the software projects of the future.

4. An introduction to mathematics on the software project

This section has a two aims First, it is a tutorial introduction to the use of mathematics on the software project. Second, it

provides a glimpse of some of the research that is being carried out into reducing the amount of proof that is required with formal methods. It describes a way of validating a design against a system specification which seems to be an improvement over previous methods — although it is still a research question as to how much an improvement can be achieved.

Before describing the mathematics it is worth stressing that I am describing one flavour of formal method known as a model-based method. There are other formal methods which are available, many of these are described in [2], however, model-based techniques have had the most industrial penetration.

4.1 The example

The example that I shall use is small, however it is rich enough to illustrate many of the principles of formal software development and some of the problems. It also represents a realistic piece of software which is used in a variety of systems.

The example is a symbol table handler. A symbol table is a collection of items which are stored and maintained in a computer system. Normally a symbol table will contain no duplicates and will have items added and removed from it during the operation of a system. Symbol tables are used everywhere in computing, for example, they are used in communications systems to keep track of calls, they are used in personnel systems to hold the names of staff employed by a company and they are used in computer operating systems to keep track of the users of the system.

I shall make a number of assumptions in writing down a specification of the symbol table:

- That four operations are required: an operation that adds an item to the symbol table, an operation that removes an item from a symbol table, an operation that returns with the number of items in the table, and an operation which checks that an identifier is stored in the symbol table.
- That the items in a symbol table will be called *identifiers*.
- That no more than $MaxIds$ identifiers are allowed in a symbol table.

- That no duplicates are allowed in the symbol table.

4.2 The specification

In writing down a model-based formal specification of the symbol table three pieces of mathematics are needed: *the state* a description of the stored data of the symbol table, a *data invariant* a predicate which describes the invariant properties of the state, and the four operations on the state.

The state is very simple. Since the only property of the symbol table is that no duplicates are allowed then a set can model the symbol table

$$SymTable : \mathbb{P}\ identifiers$$

where $\mathbb{P}$ is the power set operator. All this states is that $SymTable$ will be a set which contains identifiers. The data invariant is also quite simple. The only property that can be referred to in the data invariant is that the symbol table will contain no more than $MaxIds$ identifiers.

$$\#SymTable \leq MaxIds$$

where $\#$ is the set cardinality operator. The four operations are described by a pre-condition and a post-condition. A *pre-condition* is a predicate which must be true for an operation to be defined. A *post-condition* is a predicate which describes what happens when an operation is completed. An example of the use of these predicates is shown below in the specification of the operation $AddIdent$ which adds an identifier to the symbol table.

$$AddIdent(s : identifiers)$$
$$\text{pre}\quad s \notin SymTable \wedge \#SymTable < MaxIds$$
$$\text{post}\quad SymTable' = SymTable \cup \{s\}$$

The pre-condition states that the identifier s which is to be added to the symbol table must not already be in the table, and that there is room for the identifier in the table. The post-condition

shows the addition of s to the table, $SymTable'$ stands for the value of the symbol table after the operation has been completed. This, then, is the formal specification of the *AddIdent* operator in terms of mathematical structures which can be reasoned about. The specifications for the remaining operations are shown below, *RemoveIdent* removes an identifier from the symbol table, *NumIdent* returns with the number of items in the symbol table and *InTable* returns true if an identifier is in the symbol table.

$RemoveIdent(s : identifiers)$
pre $s \in SymTable$
post $SymTable' = SymTable \setminus \{s\}$

$NumIdent(s : identifiers)n : \mathbb{N}$
pre true
post $n = \#SymTable \wedge SymTable' = SymTable$

$InTable(s : identifiers)b : Boolean$
pre true
post $b \equiv s \in SymTable \wedge SymTable' = SymTable$

$\setminus$ stands for set subtraction and N is the set of natural numbers. The pre-condition for *NumIdent* is true since the operation is defined for all values of the state. The post-condition specifies that the symbol table is unaffected by the operation. The pre-condition for *InTable* is similarly true.

4.3 The development

The specification in the previous subsection represents an exact specification of a symbol table uncluttered with the noise that is so often found in industrial specifications. The next step is to transform the specification into program code. The technique I will use is known as *program calculation*. This method of development takes a specification and then uses a series of programming laws to transform that specification into program code [7]. The proponents of the method claim that it mirrors the process of algebraic manipulation used by mathematicians.

The first step in the development process is to select some computer data structure to model the symbol table. I shall use a single-dimension array with a fixed number of $MaxIds + 1$ locations. This will contain the identifiers with the last location holding a special value known as a *sentinel*. The reason for the sentinel will become clearer in the next section. This will formally be modelled by a total function $SymTableD$ which has a domain of consecutive integers from 1 to $MaxIds + 1$ and which always contains $MaxIds + 1$ elements in its range; a natural number $NumIds$ will be used to hold the number of identifiers currently represented in the state. $SymTableD$ models a single-dimensional array with bounds $1 \,..\, MaxIds+1$ that contains positive integers. The convention that I have used here is that the state which forms the design of the system is postfixed by a capital d. The data invariant for the state which is made up of both the array and $NumIds$ is

$$NumIds \leq MaxIds \wedge$$
$$\mathrm{dom}\, SymTableD = 1 \,..\, MaxIds + 1 \wedge$$
$$\#\,\mathrm{dom}\, SymTableD = MaxIds + 1$$

All the invariant states is the fact that the array will contain no more than $MaxIds$ elements, will range from 1 to $MaxIds$, and will have a fixed number of $MaxIds$ locations for identifiers.

Given this new design state how do you relate the specifications in the previous subsection to equivalent specifications in the design state? The answer is a predicate known as the *coupling invariant*. This is a predicate which characterizes the relationship between the state used in a specification and a design state. Whatever happens to the values in the specification state and the design state the coupling invariant will always hold. In the example used in this paper the coupling invariant is

$$SymTable = \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds))$$

where $\lhd$ is the domain restriction operator which forms a function by taking its first argument and restricting it to those elements which have their first element contained in the second operator.

Once the coupling invariant has been specified the next step is to use it to transform the specifications detailed in the previous subsection so that they refer to the design state. The only operation specification I shall consider is

$$
\begin{array}{l}
InTable(s : identifiers) \\
\textsf{pre} \quad \textsf{true} \\
\textsf{post} \; b \equiv s \in SymTable \land SymTable' = SymTable
\end{array}
$$

The manipulations on the other operations are roughly similar. We can use the coupling invariant to transform the post-condition to form a new operation $InTableD$ which operates on the design state.

$$
\begin{array}{l}
InTableD(s : identifiers) \\
\textsf{pre} \quad \textsf{true} \\
\textsf{post} \; b \equiv s \in \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds)) \land \\
\mathrm{ran}(SymTableD' \lhd (1 \,..\, NumIds)) \\
= \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds))
\end{array}
$$

We can now start making some more design decisions about the operation. I shall assume that the customer for the software has stated that for 98% of the time only a relatively few identifiers are examined when the $InTable$ operation is invoked. I shall assume that the program code which will eventually be produced will involve a linear search of the array $SymTableD$: the first element of the array will be examined, then the next, and so on, until either the end of the array has been reached or the identifier that is to be searched for has been found. Given this linear search strategy an efficient manipulation that can be made is that when an identifier has been found, it is moved to the first element of the array and all the remaining elements are shifted down by one. With this form of organization the most popular elements in the array for retrieval will usually be found near to the start of the array; in this way, the linear search will usually only involve a small number of elements.

In order to reflect the algorithm that will be used a number of transformations need to be applied to the post-condition of the

$InTableD$ operation, each transformation will preserve correctness. The first just uses a simple law of predicate calculus which reorganizes the equivalence.

$$
\begin{array}{l}
b \equiv s \in \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds)) \land \\
(s \in \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds)) \land \\
\mathrm{ran}(SymTableD' \lhd (1 \,..\, NumIds)) \\
= \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds))) \\
\quad \lor \\
(s \notin \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds)) \land \\
\mathrm{ran}(SymTableD' \lhd (1 \,..\, NumIds)) \\
= \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds)))
\end{array}
$$

The second disjunct in the predicate can be transformed into

$$
\begin{array}{l}
s \notin \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds)) \land \\
SymTableD' = SymTableD
\end{array}
$$

Then using a variable l which ranges in value from 1 to $NumIds$ the first disjunct can be transformed into

$$
\begin{array}{l}
s \in \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds)) \land \\
SymTableD(l) = s \land SymTableD'(1) = s \land \\
\forall i : 1 \,..\, l-1 \bullet SymTableD'(i+1) = SymTableD(i)
\end{array}
$$

since the predicate does not alter the range of $SymTableD$.

The post-condition of $InTableD$ has hence been transformed to

$$
\begin{array}{l}
b \equiv s \in \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds)) \land \\
(s \in \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds)) \land \\
SymTableD(l) = s \land SymTableD'(1) = s \land \\
\forall i : 1 \,..\, l-1 \bullet SymTableD'(i+1) = SymTableD(i)) \\
\quad \lor \\
s \notin \mathrm{ran}(SymTableD \lhd (1 \,..\, NumIds)) \land \\
SymTableD' = SymTableD
\end{array}
$$

The structure of the eventual software can now be discerned. It will consist of code which checks whether s is in the array. If s is in the array then it is moved to the front and the remainder of the array shifted back one; however if s is not in the array then the array remains unchanged.

4.4 Programming

The next stage is to transform the design specification for *InTableD* into program code. I shall use a simple programming language due to Dijkstra to express the code [3]. The structure of the program to carry out the search and possible adjustment of the array will reflect the structure of the post-condition:

Carry out search for s.
if s is in the array $\rightarrow$ adjust array
[] s is not in array $\rightarrow$ $SymTableD := SymTableD$
fi

This can be simplified to

Carry out search for s.
if s is in the array $\rightarrow$ adjust array
fi

By a process of refinement we can gradually aim towards the target of an implementation. The first part of the code: that of discovering s in the array requires a loop. It can be dealt with first. The technique that is used for this is to identify a loop invariant: a predicate which is true during the execution of the loop and which, when the loop terminates, will imply the post-condition which is required. The post-condition that we wish to satisfy is that connected with the search for s

$$b \equiv s \in \text{ran}(SymTableD \lhd (1\,..\,NumIds))$$

In order to satisfy this post-condition the first thing we do is to insert the identifier that is to be looked for in

$$SymTableD(NumIds + 1).$$

This identifier acts as a sentinel which cuts a search short. We can then develop code which establishes the post-condition

$$SymTableD(l) = s \wedge 1 \leq l \leq NumIds + 1 \wedge$$
$$\forall i : 1\,..\,l-1 \bullet SymTableD(l) \neq s$$

The code which established this post-condition will find the first occurrence of s inside the array $SymTableD$. A loop can be used for this with a loop invariant

$$1 \leq l \leq NumIds + 1 \wedge \forall i : 1\,..\,l-1 \bullet SymTableD(l) \neq s$$

The condition that has to be conjoined to the loop invariant to imply the post-condition is

$$SymTableD(l) = s$$

If we have a while loop which terminates when the condition in the while loop is false, then the loop condition is the negation of the above

$$SymTableD(l) \neq s$$

The structure of the program code now looks like

$SymTableD(NumIds + 1) := s$;
initialization for the loop
do $SymTableD(l) \neq s \rightarrow$
  loop body
od;
if s is in the array $\rightarrow$ adjust array
fi

Before the loop starts executing the loop invariant must be true. This can be achieved by initializing l to one. The loop must be driven to termination and this is achieved by having a statement $l := l + 1$ inside the loop. This code does not violate the loop invariant. Finally the predicate $b \equiv SymTableD(l) = s$ can be established by observing that since

$$l \leq NumIds \equiv s \in \text{ran}(SymTableD \lhd (1\,..\,NumIds))$$

and

$$b \equiv s \in \text{ran}(SymTableD \lhd (1\,..\,NumIds))$$

then

$b \equiv l \leq NumIds$

which can be established by the statement $b := l \leq NumIds$. The program can now be expressed as:

$SymTableD(NumIds + 1) := s;$
$l := 1;$
do $SymTableD(l) \neq s \rightarrow$
  $l := l + 1$
od;
$b := l \leq NumIds;$
if s is in the array $\rightarrow$ adjust array
fi

The incrementation of the loop does not violate the loop invariant so no more statements are required in the loop. The final part of the program code can now be derived. Since the loop invariant is true when the loop finishes we can say that the element s is in the array if b holds. This becomes the condition in the *if* statement.

The adjustment of the array requires that the post-condition

$$SymTableD(l) = s \wedge SymTableD'(1) = s \wedge$$
$$\forall i : 1 \,..\, l - 1 \bullet SymTableD'(i + 1) = SymTableD(i)$$

is established. Since the first conjunct has already been established all that is required is to satisfy the remaining two conjuncts. A loop is used for the third conjunct. A possible loop invariant involving a loop counter j is

$$1 \leq j \leq l \wedge$$
$$\forall i : 1 \,..\, j - 1 \bullet SymTableD'(l - i + 1) = SymTableD(l - i)$$

The condition which must be true in order to imply the post-condition is that $j = l$. Thus, if the loop is a while loop, the condition on the loop will be the negation $j \neq l$. The invariant is established by setting the variable j to be 1. The loop is driven to termination by incrementing j by 1. This means that

$$1 \leq j \leq l \wedge$$
$$\forall i : 1 \,..\, j \bullet SymTableD'(l - i + 1) = SymTableD(l - i)$$

$SymTableD(NumIds + 1) := s;$
$l := 1;$
do $SymTableD(l) \neq s \rightarrow$
  $l := l + 1$
od;
$b := l \leq NumIds;$
if $b \rightarrow$
  $j := 1;$
  do $j \neq l \rightarrow$
    $SymTableD(l - j + 1) := SymTableD(l - j);$
    $j := j + 1$
  od;
  $SymTable(1) := s$
fi

Figure 3: The final program

must be true for the invariant to hold after the incrementing of j. Thus, in order to re-establish the loop invariant what is required is that the statement $SymTableD(l - j + 1) := SymTableD(l - j)$ is executed. The second conjunction in the post-condition $SymTableD'(1) = s$ can be established by means of the statement $SymTable(1) := s$. Hence the code for the whole program will be that shown in Figure 3. The most obvious observation one can make about the mathematics displayed in the previous section concerns the volume. A large number of lines of set theory and predicate calculus were generated in order to derive a correct program. The nearest analogue in mathematics that I can think of is the calculation of the derivative of complicated functions from first principles. In defence a number of things can be said. First, that many of the steps that I described could be telescoped, some of the length of the development came about for didactic reasons. Second, a large number of the steps are quite simple, where it is obvious to the developer when a mistake is made. Third software tools are becoming available which enable the developer to check each step and partially automate the effort. Fourth, research on development using discrete mathematics is still in its

early stages, and the use of program calculation as a technique is still in its comparative infancy. Fifth, some very challenging algorithms have been developed using the technique described above. For example, Gries [4] has described an efficient binary fraction to decimal conversion routine, Morris has described the derivation of a pattern matching algorithm [8], and van Gasteren has described the development of a space efficient cyclic permutation algorithm [10]. Kaldewaij has collected together a number of program calculations in an advanced undergraduate textbook [5]. A comparison of the technique described in this paper and other mathematical development methods can be found in [6].

5. Advances and Retreats

It is clear that there are still many years to go before mathematical methods of software development will be used on even a relatively small proportion of our projects. However, they offer the hope of software with a very low level of faults and also offer tantalizing research problems to both the computer scientist and the mathematician. There have been many advances:

- There are now well-designed notations such as VDM [1] which are able to describe industrial software systems.
- Formal methods of software development have a secure place in the syllabus of the vast majority of British university computing degree courses.
- Some software development areas such as the safety-critical area are now beginning to realize the potential of formal methods of software development.
- The last three years has seen some excellent teaching books produced, for example [9].

However, to balance these advances there are a number of failures or areas where advance has been painfully slow:

- The penetration of formal methods of software development in the computing industry is minimal. I would estimate it as less than 1%.

- There is a lack of tools for software developers who use PC level computers. Those tools that have been developed are mainly confined to very powerful workstations.
- The tools that are available are experimental, and tend not to scale up to industrial size systems. Many such tools tend to be not very powerful theorem provers.
- Formal design is still a void. One solution has been described in this paper. However, although it seems to offer quite an improvement over current formal design techniques, it still requires quite a large amount of rather shallow mathematics to be generated.
- There is still a lack of integration of formal methods with other activities on the software project. For example, we know very little about the development of system tests from formal specifications.

References

[1] D. Andrews and D. Ince, Practical Formal Methods with VDM. McGraw-Hill, 1991.

[2] B. Cohen, W. T. Harwood and M. I. Jackson, The Specification of Complex Systems. Addison-Wesley, 1986.

[3] E. W. Dijkstra, A Discipline of Programming. Prentice-Hall, 1976.

[4] D. Gries, *Binary to decimal, one more time* in Beauty is our Busines., W. H. J. Feijen et al. (ed.), Springer Verlag, 1990.

[5] A. Kaldewaij, Programming: the Derivation of Algorithms. Prentice-Hall, 1990.

[6] H. J. Littek and P. J. L. Wallis, *Refinement methods and refinement calculi*, Software Engineering Journal (1992), 219–229.

[7] C. Morgan, Programming from Specifications. Prentice-Hall, 1990.

[8] J. M. Morris, *Programming by expression refinement* in Beauty is our Business, W. H. J. Feijen et al. (ed.), Springer Verlag, 1990.

[9] B. Potter, J. Sinclair and D. Till, Introduction to Formal Specification and Z. Prentice-Hall, 1991.

[10] A. J. M. van Gasteren, *Experimenting with a refinement calculus* in Beauty is our Business, W. H. J. Feijen et al. (ed.), Springer Verlag, 1990.

D. C. Ince,
Department of Computing,
The Open University,
Walton Hall,
Milton Keynes MK7 6AA,
England.

NUMERICAL SOLUTION OF CONVECTION-DIFFUSION PROBLEMS

Martin Stynes

Abstract: An overview is given of the nature of convection-diffusion problems and of some methods commonly used to solve these problems.

1. Introduction

Think of a still pond. At a point in this pond you pour a small amount of liquid dye. Approximately what shape will the dye stain take on the surface of the water as time passes? I think that we would all agree that the answer is a disc of slowly increasing radius, as the dye *diffuses* outwards from the initial point.

Consider next a more complicated situation: suppose that I replace the still pond above by a river which is flowing strongly and smoothly. What now is the shape of the dye stain?

The answer is a long thin curved wedge. This shape is the result of two physical processes: there is as before a tendency for the dye to diffuse slowly through the water, but the dominant mechanism present is the swift movement of the water, which rapidly sweeps (this is *convection*) the dye downstream. Convection alone would carry the dye along a (one-dimensional) curve on the surface; diffusion gradually spreads that curve, resulting in a wedge shape.

Physical situations such as this, where convection and diffusion are both present but convection dominates, are known

This article is the text of an invited lecture given by the author at the September meeting of the Irish Mathematical Society held at the Regional Technical College, Waterford, September 3–4, 1992.