Role of Verification in the Software Specification Process*

Marvin V. Zelkowitz
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

February 1, 1998

1 Good Software Specifications

The ability to produce correct computer programs that meet the needs of the user has been a long standing desire on the part of computer professionals. Indeed, almost every software engineering paper which discusses software quality starts off with a phrase like "Software is always delivered late, over budget and full of errors" and then proceeds to propose some new method that will change these characteristics. Unfortunately, few of these papers achieve their lofty goal. Many software systems *are* delivered late, over budget and are full of errors.

As computers become cheaper, smaller and more powerful, their spread throughout our technological society becomes more pervasive. While comic relief is often achieved by receiving an overdue notice for a bill of \$.00 (which can only be satisfied by sending a check for \$.00) or in getting a paycheck with several commas in the amount to the left of the decimal point (Alas! Such errors are quickly found!), the use of computers in real-time applications has more serious consequences.

Examples of such errors are many:

Several people have died from radiation exposure due to receiving several thousand times the recommended dosage caused by a software error in handling the backspace key of the keyboard. Only mistyping and correcting the input during certain input sequences caused the error to appear and was obviously not detected during program testing.

Several times recently entire cities lost telephone service and the national phone network was out of commission for almost 10 hours due to software errors. While not an immediate threat to life, the lack of telephone service could be one if emergency help could not be called in time.

Computers installed in automobiles since the early 1980s are moving from a passive to an active role. Previously, if the auto's computer failed, the car would still operate, but with decreased performance. Now we are entering an era where computer failure means car failure.

*Published in: *Advances in Computers* 36, Academic Press (1993) 43-109.

The increase in fly-by-wire airplanes where pilot controls only activate computers which actually control the aircraft are a potential danger source. No longer will the pilot have direct linkages back to the wings to control the flight. There is research in drive-by-wire automobiles using some of this same technology.

If told to program an anti-lock-braking system for a car, would you guarantee financially that it worked? Would you be the first person to use it?

It is clear that the need for correct software systems is growing. While the discussion that creating such software is too complex and expensive, the correct reply is that there is no other choice – we must improve the process. And, as has been demonstrated many times, it often does **not** require increased time or cost to do so!

What is a correct software system?

You probably have an intuitive meaning of the word "correct" that appeared several times already. The simple answer is that the program does what it is supposed to do. But what is that? In order to understand what its purpose, we need a description of what software should do. Right now we will informally call it the *specification.* A program is correct if it meets its specification.

It generally has been assumed that calling a program correct and stating that it meets its specification are equivalent statements. However, meeting a specification is more than that. Correctness has to do with correct functionality – given input data, does the program produce the right output? What about the cost to produce the software? The time to produce it? Its execution speed? All are attributes that affect the development process, yet the term "correctness" generally only applies to the notion of correct functionality. We will use the more explicit term *verification* to refer to correct functionality to avoid the ambiguity inherent in the term "correctness."

A simple example demonstrates that system design goes beyond correct functionality. Assume as a software manager you have two options:

1. You can build a system with 10 people in one year for $500,000.
2. You can build a system with 3 people in two years for $300,000.

Assuming both systems produce the same functionality, which do you build?

Correctness (or verification) doesn't help here. If you are in a rapidly changing business and a year's delay means you have lost entry to the market, then option 1 may be the only way to go. On the other hand, if you have an existing system that will be overloaded in 2 to 3 years and eventually need to replace it, then option 2 may seem more appropriate.

We will use the term *formal methods* to describe methods which can help in making the above decision. Verification and the quality of the resulting program is certainly a major part of our decision making and development process. However, we also need to consider techniques that address other attributes. Some of these other attributes include:

Safety. Can the system fail with resulting loss of life? With the growth of real-time computer-controlled systems, this is becoming increasingly important. Techniques, such as *software fault tolerance*, are available for analyzing such systems.

Performance. Will the system respond in a reasonable period of time? Will it process the quantity of data needed?

Reliability. This refers to the probability that this system will exhibit correct behavior. Will it exhibit reasonable behavior given data not within its specification? A system that crashes with incorrect data might be correct (i.e., always produces the correct output for valid input) but highly unreliable.

Security. Can unauthorized users gain information from a software system that they are not entitled to? The privacy issues of a software system also need to be addressed.

Resource utilization. How much will a system cost? How long will it take? What development model is best to use?

This paper presents a brief survey of verious verification methods necessary to show correct functionality of a software product. However, as briefly shown already, verification is only one aspect of the problem. We later address some of these other important aspects in producing good software specifications. More detailed discussion of other formal methods can be found elsewhere (e.g., (Craigen, 1990) (Wing, 1990)). In the rest of this section we will briefly discuss the role of verification from an historical prospective, and then briefly summarize the techniques we will address more fully later.

1.1 An Historical Prospective

The modern programming language had its beginnings in the late 1950s with the introduction of FORTRAN, Algol-60 and LISP. During this same period, the concepts of Backus-Naur Form (BNF), context free languages and machine automated syntax analysis using these concepts (e.g., parsing) were all being developed.

Building a correct program and elimination of "bugs[1]" was of concern from the very start of this period. It was initially believed that by giving a formal definition of the syntax of a language, you would eliminate most of these problems.

1.1.1 Syntax

By *syntax*, we mean what the program looked like. Since this early period, BNF has become a standard model for describing how the components of a program are put together. Syntax does much for describing the meaning of a program. Some examples:

Sequential execution. Rules of the form:

$$< stmtlist > \rightarrow < stmt >; < stmtlist > \; | \; < stmt >$$

does much to convey the fact that execution proceeds from the first $< stmt >$ to the remaining statements.

Precedence of expressions. The meaning of the expression $2 + 3 \times 4$ to be $2 + (3 \times 4) \; = \; 14$ and not $(2 + 3) \times 4 \; = \; 20$ is conveyed by rules like:

$$\begin{array}{l} < expr > \rightarrow < expr > + < term > \; | \; < term > \\ < term > \rightarrow < term > \times < factor > \; | \; < factor > \end{array}$$

[1] A term Grace Hopper is said to have coined in the late 1950s after finding a bug (insect variety) causing the problem in her card reading equipment.

In this case, $\times$ has "higher" precedence than $+$.

However, there is much that syntax cannot do. If a variable was used in an expression, was it declared previously? In a language with block structure like Ada or Pascal which allows for multiple declarations of a variable, which declaration governs a particular instance of an identifier? How does one pass arguments to parameters in subroutines? Evaluate the argument once on entry to the subroutine (e.g., *call by value*) or each time it is referenced (e.g., *call by reference*, *call by name*)? Where is the storage to the parameter kept?

All these important issues and many others cannot be solved by simple syntax. So the concept of programming *semantics* (i.e., what the program means) developed.

1.1.2 Testing

Time for a slight digression. What is wrong with program testing? Software developers have been testing and delivering programs for over 40 years.

The examples cited at the beginning clearly answer this question. Programs are still delivered with errors, although the process is slowly improving.

As Dijkstra has said "Testing shows the presence of errors, not their absence." He gives a graphic demonstration of the failure of testing. To test the correctness of $A + B$ for 32-bit integers A and B, you need to perform $2^{32} \times 2^{32} \sim 10^{20}$ tests. Assuming 10^8 tests per second (about the limit in early-1992 technology), that would require more than 30,000 years of testing!

Testing is only an approximation to our needs. Verification is the important concept.

1.1.3 Attribute grammars

Probably the first semantic model was the attribute grammar of Knuth in 1968(Knuth, 1968). In this case, attributes (values) were associated with each node in the syntax tree, and this information could be passed up and down the tree to derive other properties. The technique is particularly useful for code generation in a compiler. For example, the expression evaluation problem above could be solved by the following set of attributes for producing Polish postfix for expressions:

$$\begin{array}{lll}
<expr> \rightarrow & <expr> + <term> & Postfix(expr_1) = Postfix(expr_2) || Postfix(term) || + \\
 & \mid\ <term> & Postfix(expr) = Postfix(term) \\
<term> \rightarrow & <term> \times <factor> & Postfix(term_1) = Postfix(term_2) || Postfix(factor) || \times \\
 & \mid\ <factor> & Postfix(term) = Postfix(factor)
\end{array}$$

where x_1 and x_2 refer to the left and right use, respectively, of that nonterminal in the production and $||$ is the string concatenation operator.

With these rules, it is clear that given the parse tree for the expression $2 + 3 \times 4$, its correct postfix is $2\ 3\ 4\ \times\ +$ yielding the value 14.

While useful as a technique in compiler design, attribute grammars do not provide a sufficiently concise and formal description of what we have informally called the specification of a program.

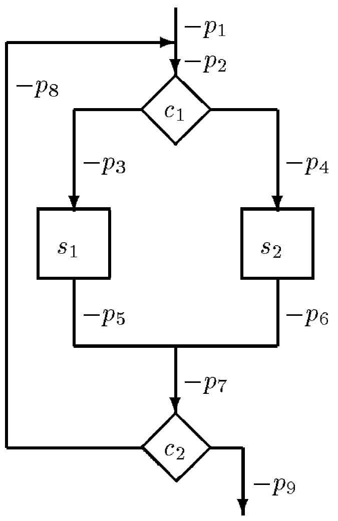

Figure 1: Floyd verification model

1.1.4 Program verification

The modern concept of program verification was first introduced by Floyd in 1967 (Floyd, 1967). Given the flowchart of a program, associate with each arc a predicate that must be true at that point in the program. Verification then consisted of proving that given the truth of a predicate before a program node, if that node was executed, then the predicate following the node would be true. If this could be proven for each node of the flowchart, then the internal consistency of the entire program could be proven. We would call the predicate associated with the input arc to the program the input condition or *precondition*, the predicate on the output arc to the program the output condition or *postcondition* and the pair would be our *specification* since they determined the input/output behavior of the program.

For example, Figure 1 contains four program nodes: c_1, c_2, s_1 and s_2 and nine predicates: $p_1, \ldots, p_9$. In order to show the correctness of this program, we have to prove the following propositions:

$$\begin{array}{ll}
p_1 \lor p_8 \Rightarrow p_2 & p_2 \land c_1 \Rightarrow p_3 \\
p_2 \land \neg c_1 \Rightarrow p_4 & p_3 \textit{ and } s_1 \textit{ executes} \Rightarrow p_5 \\
p_4 \textit{ and } s_2 \textit{ executes} \Rightarrow p_6 & p_5 \lor p_6 \Rightarrow p_7 \\
p_7 \land c_2 \Rightarrow p_8 & p_7 \land \neg c_2 \Rightarrow p_9
\end{array}$$

Once we have determined the effects of statements s_1 and s_2, all, except one, of these propositions are fairly straightforward. The one problem is the very first:

$$p_1 \lor p_8 \Rightarrow p_2$$

Since p_2 depends upon p_8, and developing p_8 depends upon all the previous p_is, the process of generating predicates for loops in the program becomes very difficult. It would take a further development by Hoare to fix this problem.

In 1969 Hoare introduced the axiomatic model for program verification which put Floyd's model into the formal context of predicate logic (Hoare, 1969). His basic approach was to extend our formal mathematical theory of predicate logic with programming language concepts. His basic notation was: $\{P\}S\{Q\}$ meaning that if P were the precondition before the execution of a statement S, and if S were executed, then post-condition Q would be true. Since a program is a sequence of statements, we simply needed a set of axioms to describe the behavior of each statement type and a mechanism for executing statements sequentially. As will be shown later, this model simplifies (but does not eliminate) the problems with loop predicates as given in the Floyd model.

1.2 A brief survey of techniques

Since the late 1960s and the developments of Floyd and Hoare, several models for program verification have been developed. These can be broken down into three general categories: (1) An axiomatic model as an extension to predicate logic and mathematical theorem proving; (2) A functional model which views programs as functions from some input space to some output space; and (3) An algebraic model which models data interactions.

These three general categories can be divided into five verification techniques, which will be breifly described below. We will explore three of these, the axiomatic, functional and denotational semantics models, in greater detail later in this paper.

Throughout this paper we will model the same simple language using the various techniques. The syntax of this language is:

$$\begin{aligned} <stmt>::= \; & <stmt>;<stmt> \\ & |\mathbf{begin}\ <stmt>\ \mathbf{end} \\ & |\mathbf{if}\ <expr>\ \mathbf{then}\ <stmt> \\ & |\mathbf{if}\ <expr>\ \mathbf{then}\ <stmt>\ \mathbf{else}\ <stmt> \\ & |\mathbf{while}\ <expr>\ \mathbf{do}\ <stmt> \\ & |<id> \leftarrow <expr> \end{aligned}$$

where $< id >$ and $< expr >$ have their usual meanings.

1.2.1 Axiomatic verification

This is the technique as previously described by Floyd and Hoare. We can give a series of axioms describing the behavior of each statement type, and prove, using formal mathematical logic, that the program has the desired pre- and postconditions.

For example, given two propositions: $\{P\}S_1\{Q\}$ and $\{Q\}S_2\{R\}$, we can infer from this that if we execute both statements, we get: $\{P\}S_1;S_2\{R\}$. Similarly if we can prove the following proposition: $R \Rightarrow T$, we can then state: $\{P\}S_1;S_2\{T\}$. Continuing in this manner, we build up a proof for the entire program. We can extend this model to include data declarations, arrays and procedure invocation.

1.2.2 Predicate transforms

Dijkstra developed a model similar to Hoare's axiomatic model which he called *predicate transforms* and is based upon two notions: (1) The *weakest precondition* of a statement; and (2) Guarded commands and nondeterministic execution.

Weakest precondition. The weakest precondition to a given statement S and postcondition Q is the largest initial set of states for which S terminates and Q is true. If P is the weakest precondition, we write $P = wp(S, Q)$.

From this basic definition we can prove several theorems:

$$\begin{aligned} wp(S, false) &= false \\ if\ P \Rightarrow Q\ then\ wp(S,P) &\Rightarrow wp(S,Q) \\ wp(S, P \vee Q) &= wp(S,P) \vee wp(S,Q) \\ wp(S, P \wedge Q) &= wp(S,P) \wedge wp(S,Q) \end{aligned}$$

Guarded commands. Dijkstra realized that many algorithms were easier to write nondeterministically, i.e., "if this condition is true then do this result." A program is simply a collection of these operations, and whenever any of these conditions apply, the operation occurs. This concept is also the basis for Prolog execution.

The basic concept is the guard ($[\!]$). The statement:

$$\mathbf{if}\ a_1 \rightarrow b_1\ [\!] \cdots [\!]\ a_n \rightarrow b_n\ \mathbf{fi}$$

means to execute any b_i if the corresponding a_i is true. Execution may be nondeterministic (but still sequential) if more than one guard is true.

From these, we can build up a model very similar to the Hoare axioms. For more information on this technique, the text by Gries (Gries, 1981) or Dijkstra's original paper (Dijkstra, 1975) are good sources. Basu and Yeh provide important modifications to the original definitions (Basu, 1975).

1.2.3 Algebraic specification

The use of modularization, datatypes, and object oriented programming have lead to a further model called *algebraic specifications* as developed by Guttag (Guttag, 1980). In this model we are more concerned about the behavior of objects defined by programs rather than the details of their implementation. For example, "What defines a data structure called a *stack*?" Any such description invariably includes the example of taking trays off and on a pile of such trays in a cafeteria and moving your hands up and down. More formally, we are saying that a *push* of a tray onto the stack is the inverse of the *pop* operation taking it off the stack. Or in other words, if S is a stack and x is an object, *push* and *pop* obey the relationship

$$pop(push(S, x)) = S$$

That is, if you add x to stack S and then pop it off, you get S back again.

By adding a few additional relationships, we can formally define how a stack must behave. We do not care how it is implemented as long as the operations of *push* and *pop* obey these relationships. That is, these relationships now form a *specification* for a stack. In addition to (Guttag, 1980), the paper by Guttag and Horning (Guttag, 1978) provides much of the mathematical background for this theory. The use of term-rewriting systems and the Knuth-Bendix algorithm is a significant part of the proof methodology used to determine correctness of an algebraic specification. Two papers by Musser provide a good introduction to this model by describing the implementation of his AFFIRM system, an implementation on this proof technique (Musser, 1979) (Musser, 1980).

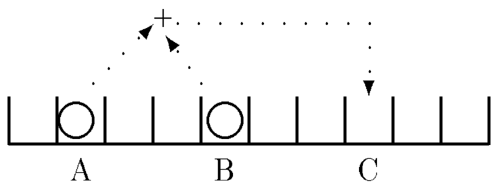

Figure 2: Memory model of storage

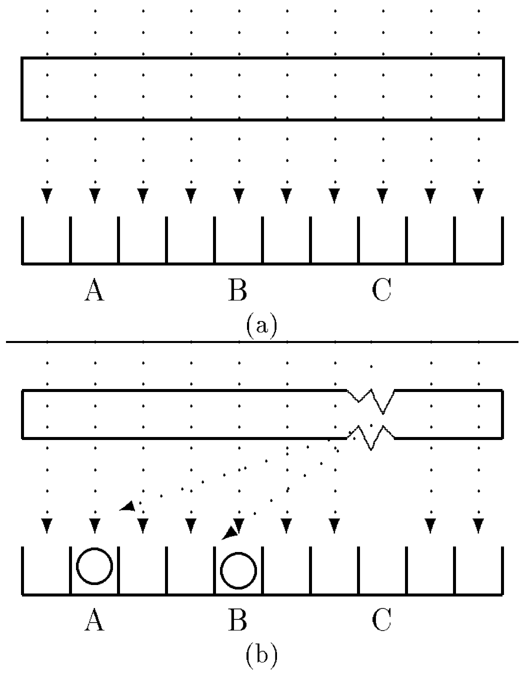

Figure 3: Applicative assignment

Storage

Before discussing the remaining techniques, a slight digression concerning assignment and memory storage is in order. Consider the following statement: $C \leftarrow A+B$. This statement contains two classes of components: (1) A set of data objects $\{A, B, C\}$ and (2) operators $\{+, \leftarrow\}$.

In most common languages like FORTRAN, Pascal, C, or Ada, operators are fixed and changes are made to data. Storage is viewed as a sequence of cells containing "memory objects." The various operators access these memory objects, change them, (e.g., like accessing A and B and adding them together) and placing the resulting object in the location for C (Figure 2). An ordered collection of colored marbles is probably the mental image most people have of memory.

On the other hand, we can view data as fixed and manipulate the operator that views the objects (Figure 3(a)). In this case we model an operator as a lens that allows us to see the corrected data as modified by the given statement. We call this the applicative view of storage.

In this case, memory is simply a function that returns a value for each location. Execution of an assignment statement simply modifies the accessing function (Figure 3(b)). In other words, the modified function is simply the composition of the original function and the new statement.

Pure applicative programming is the basis for languages like LISP; however, most LISP implementations do include memory storage features. As we shall see, this model is also the basis for the denotational and functional models to be next described.

1.2.4 Functional correctness

A program can be considered as a function from some input domain to some output domain. If we can also represent a specification as a function, we simply have to show that they are equivalent functions.

Using the box notation of Mills (Mills, 1987a), if p is a program, then $\boxed{p}$ is the function that the program produces. If f is the specification of this program, then verification means showing the equivalence of $f = \boxed{p}$. While this is in general an undecidable property, we can develop conditions of certain programs where we can show this.

In particular, if a program p is a sequence of statements $s_1; \cdots, s_n$, then $\boxed{p}$ is just the functional composition of each individual statement function $\boxed{s_1} \circ \cdots \circ \boxed{s_n}$.

We later give several axioms for deriving statement functions from programming language statements and present techniques for proving this equivalence.

1.2.5 Operational semantics

The final model of verification we shall discuss is the operational model. In this case, model the program using some higher level interpreter, show that the program and the abstracted program have equivalent properties, and "execute" the program in the abstract model. Whatever effects the abstracted program show will be reflected in the concrete program.

The first such model based upon this technique was the Vienna Definition Language (VDL[2] (Lucas, 1969, Marcotty, 1976)). This model extended the parse tree of a program into a "tree interpreter." (See Figure 4). While the parse tree (component *s_tree*) was a static component of this model, some components like the program store (i.e., data storage component *s_data*) and internal control (*s_control*) were dynamic and changed over time and other components like the library (i.e., "microprogrammed" semantic definition of each statement type in *s_library*) were also static. The semantic definition of the language was the set of interpreter routines built into the library routines. A LISP-like notation was used to define the semantic routines.

VDL was used briefly in the 1970s, and was used as the basis for the standardized definition for the language PL/I, but is largely obsolete today. The major problem is that "debugging" the interpreter routines in the control aspect of the model is not very different from debugging programming language statements. While the model did present the program's semantics at a higher more abstract level, the level was not deemed high enough and the information obtained not strong enough to warrant its use.

A second operational model is still very much in use and has had major impact on the other models and upon practical specification systems. That is the technique of *denotational semantics* developed by Scott

[2]Not to be confused with the Vienna Development Method or VDM to be described later.

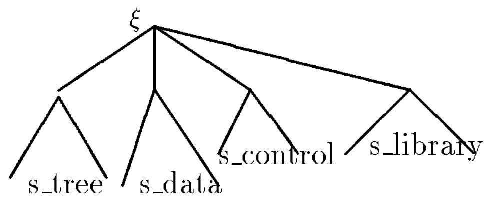

Figure 4: Vienna Definition Language Model

and Strachey (Scott, 1971). In this case, similar to the functional model of Mills, we view a program as function from one domain to another.

A fundamental idea is that we view "memory" as simply a function from a set of identifiers to a set of values. Thus the *state* of a computation at any time is simply a function m_i with the functional signature $id \rightarrow value$ meaning for each $j \in id, m_i(j) = k,\ k \in value$. The "execution" of statement s_{i+1} simply means we transform function m_i into function m_{i+1} by the composition with the function for statement s_{i+1}. (Note the similarity with the Mills functional approach above.)

We give a formal description of the functionality that each statement has on the underlying program store to produce a new program store, and can use this to model complete languages.

This model has had an effect on other models. The basic functional approach is the basis for the Mills' functional model. We will later see that the axiomatic array axiom is just the denotational semantics assignment property.

1.3 Semantics versus specifications

In the discussion so far, the terms "semantics," "verification," and "(functional) specification" have been mostly intermixed with no clear distinction among them. They are highly interrelated, generally describe similar properties, and their order above generally follows the historical development of the field.

Initially (late 1950s through mid-1960s), the problem was to describe the semantics of a programming language, using techniques like attribute grammars and VDL-like operational semantics. The thrust through the 1970s was the proving of the (functional) correctness of a program, or program verification. Today, we are interested in building valid systems, i.e., programs that meet their specifications.

Using the functional correctness box notation, we can describe some of today's issues. Let s be a specification and p be a program.

- Does $s = \boxed{p}$? This is the verification problem.
- Given s, build program p such that $s = \boxed{p}$. This is the program design problem.
- Given p, find s such that $s = \boxed{p}$. This is what we call *reverse engineering* today. It is an important component in the interest in software reuse – given a program module, what function does it implement? If we can answer this – semiautomatically – we can more easily reuse the module and reduce development costs.

While all these problems are different, the techniques to address them are similar. Therefore, a study of verification issues has major impact on all of the semantic, verification, specification and reverse engineering problems.

1.4 Limitations of formal systems

Now that you are "sold" on the value of such formal models, we must put these techniques in perspective. Hall (Hall, 1990) listed 7 "myths" of formal systems. It is important to understand these concepts as part of learning about the techniques themselves.

1. *Formal methods can guarantee that software is perfect.* As we have shown, all the formal techniques rely on abstracting a program into an abstract specification that closely approximates reality. However, this formal specification is rarely exact, so the resulting program only approximates what you specify. If done well, then this approximation is close enough. However, even simple propositions like "for integer x, $x + 1 > x$" fail with real machines with fixed word size and limited range of integer values.

   In addition, we cannot forget that mathematical proofs may have errors in them. Formal proofs certainly help, but are no guarantee of perfection.

2. *They work by proving the programs are correct.* As stated at the beginning, we are interested in more than just functional correctness. Cost, development time, performance, security, safety, etc. are all properties that are part of a complete specification.

3. *Only highly critical systems benefit from their use.* It has been shown that almost any large system will benefit from using formal techniques. The *cleanroom* (Mills, 1987b) is a technique to informally use functional correctness on large software projects. It has been used at IBM, NASA Godddard Space Flight Center and at the University of Maryland (albeit with small student projects in this case). In all instances, reliability was higher and development effort was sometimes dramatically lower.

4. *They involve complex mathematics.* These techniques involve *precise* mathematics, not complex mathematics. There is nothing written here that a well-informed college undergraduate should not be able to understand. Precision takes much of the ambiguity out of a 300 page informal English specification that is more typical in industry today.

5. *They increase the cost of development.* They do increase the cost of program design, since one must develop the abstract model of the specification more explicitly that is usually done. For managers impatient with programmers not writing any "code" this certainly looks like an increase in costs. But as numerous studies have shown (e.g., the cleanroom studies above), the overall costs are lower since the time consuming and expensive testing phases are dramatically reduced.

6. *They are incomprehensive to clients.* It is our belief that a 300 page English text of ambiguous statements is more incomprehensible. In addition, the role of formal methods is to help the technical staff understand what they are to build. One must still translate this into a description for the eventual user of the system.

7. *Nobody uses them for real projects.* This is simply not true. Several companies depend upon formal methods. Also, various modifications to some of these techniques are used in scattered projects throughout industry. It is unfortunately true, however, that few projects use such techniques, which results in most of the problems everyone is writing about.

2 Axiomatic correctness

An axiomatic approach requires both axioms and rules of inference. For our purposes, we will assume the axioms of arithmetic which characterize the integers and the usual rules of logical inference. However, we need additional axioms and inference rules corresponding to the constructs we intend to use as our programming language.

2.1 Programming Language Axioms

In order to build programs into our inference system, we must be able to model a computation as a logical predicate. We use the notation $\{P\}S\{Q\}$, where P and Q are assertions about the state of the computation and S is a statement. This expression is interpreted as "If P, called the *precondition*, is true before executing S and S terminates normally, then Q, called the *postcondition*, will be true." This will allow us to model a simple "algol like" programming language. We do this by adding the inference rules of composition and consequence:

$$\textbf{Composition}: \quad \frac{\{P\}S_1\{Q\},\ \{Q\}S_2\{R\}}{\{P\}S_1;S_2\{R\}}$$

$$\textbf{Consequence}_1: \quad \frac{\{P\}S\{R\},\quad R \Rightarrow Q}{\{P\}S\{Q\}}$$

$$\textbf{Consequence}_2: \quad \frac{P \Rightarrow R,\quad \{R\}S\{Q\}}{\{P\}S\{Q\}}$$

These rules are interpreted as "If we know that the antecedent (expressions above the line) is true, then we can infer that the consequent (expression below the line) follows."

The rule of composition is the basic mechanism which allows us to "glue" together two computations, i.e., two statements. If the postcondition of one statement is the same as the precondition of the following statement, then the execution of both of them proceeds as expected.

The rules of consequence builds in our predicate logic rules of inference. We can use these to permit us to use alternative pre- and postconditions to any statement as long as we can derive them via our mathematical system.

The basic approach is a "backwards verification" method. For example, for the assignment, we will define the axiom so that given the result of an assignment statement, what must have been true prior to the execution of the statement in order for the result to occur? Or in other words, given the postcondition to an assignment statement, what is its precondition?

For assignment we will use the *schema*

$$\{P^x_y\}x \leftarrow y\{P\}$$

Here P^x_y represents the expression P where all free occurrences of x have been replaced by y.

In addition to the above axiom schema, the following rules of inference define the **if** and **while** statements:

$$\textbf{Conditional}_1: \quad \frac{\{P \wedge B\}\ S\ \{Q\},\quad P \wedge \neg B \Rightarrow Q}{\{P\}\textbf{if } B \textbf{ then } S\{Q\}}$$

$$\textbf{Conditional}_2: \quad \frac{\{P \wedge B\}\ S_1\ \{Q\},\quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\ \{Q\}}$$

$$\textbf{While}: \quad \frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\textbf{while } B \textbf{ do } S\ \{P \wedge \neg B\}}$$

The **if** statement rules are fairly obvious. Each of the two possible rules simply traces the execution of the corresponding $if - then$ and $if - then - else$ statement, and given precondition P, determines under what conditions will postcondition Q be true.

The axiom for the **while** statement is perhaps as major a contribution by Hoare as the entire model of verification that he also developed. The basic loop has the structure:

$$\begin{array}{l} \textbf{while } something - is - true \\ \qquad\qquad \textbf{do } something \end{array}$$

There must be some property that remains true throughout the execution of the loop. So if we call the body of the loop S, then we will call the property that must remain true P, and we have the result: "If P is true and we execute the loop then P will remain true." The condition 'and we execute the loop' is just the predicate on the **while** statement B. This leads to the condition that: $\{P \wedge B\}S\{P\}$ as the antecedent property on the **while** axiom. So if this antecedent is true, then after the loop 'terminates,' we still will have P true, and since the loop terminated, B must now be false, hence the axiom as given. Note we have not proven that the loop *does* terminate. That must be shown and is outside of this axiom system.

The property that remains true within the loop is called the *invariant* and is at the heart of developing axiomatic proofs for programs.

2.1.1 Program Termination

The preceding axioms show the "partial correctness" of a program; that is, if the program begins execution in a state satisfying the precondition, and if the program terminates, then the the postcondition will be true. But how do we know the program actually terminates? We can often show termination by showing that the following two properties hold:

1. Show that there is some property P which is positive within a loop.
2. Show that for each iteration of the loop, P is decremented by a fixed amount. That is $P_i > P_{i+1}$.

If both properties are true, and if the second property causes P to decrement (yet still be positive), then the only way this can be consistent is for the loop to terminate, else the first property must become false at some point.

2.1.2 Sample axiomatic proof

We will now demonstrate the axiomatic technique by proving the correctness of the following program which computes the product of A and B:

```
        {$B \geq 0$}
1.      $MULT(A, B) \equiv$
2.          $a \leftarrow A$
3.          $b \leftarrow B$
4.          $y \leftarrow 0$
5.          while $b > 0$ do
6.              begin
7.              $y \leftarrow y + a$
8.              $b \leftarrow b - 1$
9.              end
        {$y = AB$}
```

The general approach is to work backwards starting with the postcondition to show that the precondition yields that postcondition. In this case, the program terminates with a **while** statement (lines 5-9). Therefore we need to determine the invariant for the loop. If I is the invariant and X is the **while** statement conditional, then we have to show the following two properties:

a. $\{I \wedge X\}$ *lines* $7 - 8\{I\}$
b. $(I \wedge \neg X) \Rightarrow (y = AB)$

The first property will allow us to build the **while** statement, while the second property shows that we get the multiplication answer that we desire.

In order to develop the invariant I, it is necessary to determine what property doesn't change within the loop. The goal of the loop is to compute AB, which is a constant. Since the loop is increasing y while decreasing b, combining them gives a clue to our invariant. Since ab is just AB initially, we get a clue that our invariant might be something like:

$$y + ab = AB$$

Since we want $b = 0$ at the end, forcing y to be equal to AB, our invariant becomes:

$$I \equiv (y + ab = AB) \ \wedge \ (b \geq 0)$$

Since X is just $b > 0$, $I \wedge \neg X \Rightarrow (y = AB)$ is shown as follows:

$$\begin{array}{l} I \wedge \neg X \equiv \\ (y + ab = AB) \ \wedge \ (b \geq 0) \ \wedge \ \neg(b > 0) \equiv \\ (y + ab = AB) \ \wedge \ (b \geq 0) \ \wedge \ (b \leq 0) \equiv \\ (y + ab = AB) \ \wedge \ (b = 0) \equiv \\ (y + a0 = AB) \equiv \\ (y = AB) \end{array}$$

In order to show that I is the invariant of the loop:

Line 8. $b \leftarrow b - 1$

$$\{y + a(b-1) = AB \wedge (b-1) \geq 0\} b \leftarrow b - 1 \{y + ab = AB \wedge b \geq 0\} \quad \textit{Assignment axiom}$$

Line 7. $y \leftarrow y + a$

$$\begin{array}{ll}
\{y + a + a(b-1) = AB \wedge b - 1 \geq 0\} y \leftarrow y + a \{y + a(b-1) = AB \wedge b - 1 \geq 0\} & \textit{Assignment axiom} \\
y + ab = AB \Rightarrow y + a + a(b-1) = AB & \textit{Mathematical theorem} \\
y + ab = AB \Rightarrow (y + a) + a(b-1) = AB, & \textit{Rule of Consequence} \\
\underline{\{y + a + a(b-1) = AB \wedge b - 1 \geq 0\} y \leftarrow y + a \{y + a(b-1) = AB \wedge b - 1 \geq 0\}} & \\
\{y = ab = AB \wedge b - 1 \geq 0\} y \leftarrow y + a \{y + a(b-1) = AB \wedge b - 1 \geq 0\} &
\end{array}$$

Lines 7-8

$$\begin{array}{ll}
\{y + ab = AB \wedge b - 1 \geq 0\} y \leftarrow y + a \{y + a(b-1) = AB \wedge b - 1 \geq 0\}, & \textit{Rule of composition} \\
\underline{\{y + a(b-1) = AB \wedge b - 1 \geq 0\} b \leftarrow b - 1 \{y + ab = AB \wedge b \geq 0\}} & \\
\{y + ab = AB \wedge b - 1 \geq 0\} y \leftarrow y + a; b \leftarrow b - 1 \{y + ab = AB \wedge b \geq 0\} &
\end{array}$$

We get the loop invariant as follows:

$$\begin{array}{l}
I \wedge X \equiv \\
(y + ab = AB) \wedge (b \geq 0) \wedge (b > 0) \Rightarrow \\
(y + ab = AB) \wedge (b > 0) \Rightarrow \\
(y + ab = AB) \wedge b - 1 \geq 0
\end{array}$$

By the rule of consequence we get:

$$\{y + ab = AB \wedge b > 0\} y \leftarrow y + a; b \leftarrow b - 1 \{y + ab = AB \wedge b \geq 0\}$$

The above is just $\{I \wedge X\} lines\ 7 - 8 \{I\}$. Therefore we can use our **while** axiom:

$$\frac{\{y + ab = AB \wedge b > 0\} y \leftarrow y + a; b \leftarrow b - 1 \{y + ab = AB \wedge b \geq 0\}}{\{y + ab = AB \wedge b \geq 0\} \textbf{while} \cdots \textbf{end} \{y + ab = AB \wedge b \geq 0 \wedge \neg b > 0\}}$$

We have already shown that this results in $y = AB$. We finish up by going backwards through the initial assignment showing that the program's precondition yields this invariant:

$$\begin{array}{ll}
\{0 + ab = AB \wedge b \geq 0\} y \leftarrow 0 \{y + ab = AB \wedge b \geq 0\} & \textit{Line } 4 \\
\{0 + aB = AB \wedge B \geq 0\} b \leftarrow B \{0 + ab = AB \wedge b \geq 0\} & \textit{Line } 3 \\
\{0 + AB = AB \wedge (B \geq 0)\} a \leftarrow A \{0 + aB = AB \wedge B \geq 0\} & \textit{Line } 2
\end{array}$$

Since $B \geq 0 \Rightarrow 0 + AB = AB \wedge B \geq 0$ we get by consequence:

$$\{B \geq 0\} a \leftarrow A \{0 + aB = AB \wedge B \geq 0\}$$

Combining lines 2-4 by the rule of composition, we complete the proof:

$$\{B \geq 0\} \mathbf{MULT(A, B)} \{y = AB\}$$

Termination. We have only shown that *if* the program terminates, then it gives us the desired answer. We must also show termination.

1. Let property P be $b > 0$. P is obviously true in loop.
2. The only change to b in loop is $b \leftarrow b - 1$. So P is decremented in loop.

Since both termination properties are true, the loop must terminate, and we have show total correctness of the program.

2.2 Array Assignment Axiom Schema

Consider a use of our existing axiom of assignment as in the expression below:

$$\{?\} x[i] \leftarrow y \{x[j] = z\}$$

What must be true *before* an assignment to the array element $x[i]$ to guarantee the given equality after the assignment? If the expression only uses array entries in the right hand side of the assignment, then our old rule still works. However, we need an inference rule to handle the case where the target of an assignment is an array value.

The key is to consider array assignments as operations which change the value of an entire array rather than a single element. In this case, we will borrow techniques that will become more apparent when we present denotational semantics later. We will consider the entire set of values that every element of an array has, and view an array assignment as a function which transforms this set of values into a new set with a specific member of that set altered.

If the array x is viewed as a function that associates a value for each element i, then the assignment $x[i] \leftarrow y$ assigns to x the function like the old value of x except that it maps argument i to y. That is,

$$\alpha(x, i, y)[j] = \begin{cases} y, & when\ j = i \\ x[j], & when\ j \neq i \end{cases}$$

Hence, notationally we can discuss and manipulate the different "states" attained by our array during a program execution simply by choosing the appropriate function name. Therefore, our new axiom schema is

$$\textbf{Array assignment}: \quad \{P^x_{\alpha(x,i,y)}\} x[i] \leftarrow y \{P\}$$

We can use this new axiom with our previous rules of inference to prove properties of programs containing arrays, such as:

$$\{\alpha(x,k,7)[1]=7\}x[k] \leftarrow 7\{x[1]=7\}$$
$$\{\alpha(x,1,7)[1]=7\}k \leftarrow 1\{\alpha(x,k,7)[1]=7\}$$
$$\mathbf{true} \ \Rightarrow \ \alpha(x,1,7)[1]=7$$
$$\frac{\{\mathbf{true}\}\ k \leftarrow 1\ \{\alpha(x,k,7)[1]=7\};\{\alpha(x,k,7)[1]=7\}x[k] \leftarrow 7\{x[1]=7\}}{\{\mathbf{true}\}\ k \leftarrow 1; x[k] \leftarrow 7\{x[1]=7\}}$$

2.3 Procedure Call Inference Rules

We are now in a position to extend the language we are modeling to include declarations, blocks and procedure calls. As before, we need to add the appropriate inference rules.

In order to define scope rules and local variables, variable x inside statement S will be a local variable if it doesn't affect any pre- or postcondition around S. That is, if $\{P\}S\{Q\}$ is proven, and variable x does not appear in either P or Q, then we can make x a local variable inside S. By defining S^x_y to mean substitute all occurences of x by a variable y that doesn't appear in either P or Q, then we have the following inference rule for declarations:

$$\textbf{Declaration}: \quad \frac{\{P\}S^x_y\{Q\}}{\{P\}\ \textbf{begin new}\ x;\ S\ \textbf{end}\ \{Q\}}$$

Several rules deal directly with procedure calls which involves proving the behavior of the body of a procedure and then encapsulating that behavior in a separate procedure declaration. We need to first define the encapsulation mechanism for a procedure and then define the operation of parameter passing. First the *actual* parameter names must be installed, and then the assertions which describe a procedure's behavior must be adapted for use in the caller's context.

If we can show $\{P\}S\{Q\}$, then we should be able to execute S by the procedure name ω. The problem occurs if the parameters to S also appear in P and Q. Our rule of *invocation*, wherein we can capture a code body as a procedure, is

$$\textbf{Invocation}: \quad \frac{\omega(x):(v)\ \textbf{procedure}\ S,\ \ \{P\}S\{Q\}}{\{P\}\ \textbf{call}\ \omega(x):(v)\ \{Q\}}$$

where x and v represent *lists* of variables which can change and which do not change in the body of code S, respectively. Furthermore, we explicitly disallow the body S from ever referencing a non-local variable (hence all program variables which are used *must* appear in either x or v.) Informally, we would read this rule as "If ω is a procedure whose body is implemented by S, and if we know executing S yields Q from P, then we can infer that calling ω from state P will yield Q."

Next is our rule of *substitution* for obtaining an expression involving *actual* (instead of *formal*) parameters in a procedure call.

$$\textbf{Substitution}: \quad \frac{\{P\}\ \textbf{call}\ \omega(x):(v)\ \{Q\}}{\{P^{k\ x\ v}_{k'\ a\ e}\}\ \textbf{call}\ \omega(a):(e)\ \{Q^{k\ x\ v}_{k'\ a\ e}\}}$$

where x and v again represent the lists of formal parameters which change and do not change, respectively; a and e represent the *actual* parameters in the call; and the lists k and k' have the following special interpretation: Because we want to eliminate the case of aliasing, k is a list of all symbols which are free in P and R but do not appear in the procedure interface, and which furthermore correspond to symbols appearing in either a or e. This list is thus obtained by first enumerating all free variables in P and Q, crossing out those appearing in either x or v, and then crossing out those that do *not* appear in either a or e. Any of these symbols k which then appear in P or Q must then be replaced with completely new symbols k'; it is essential that this substitution be performed *first* in applying this substitution rule.

Except for this list of variables k as just described, the substitution rule above would be quite straight forward and intuitive: we would be simply plugging in actual parameters for the formals and then using the call.

Once we have an expression for procedure calls using the actual parameters, we are ready to take any assertions used to describe the procedure and adapt them to the context of the call. What we want is a rule similar to the previous rule of consequence, as in the following *incorrect* rule:

$$\textbf{Improper consequence}: \quad \frac{\{P\}\textbf{call}\,\omega(a):(e)\ \{R\},\quad R \Rightarrow Q}{\{P\}\textbf{call}\,\omega(a):(e)\ \{Q\}}$$

Why can't we use such a rule? The reason is that we only know the pre- and postcondition to the call to ω, we do not know the details of ω. It is possible that there are changes to the parameter a or to other free variables within P and Q. These side effects makes the above an invalid rule of inference.

Instead we need a more complex rule of *adaptation*, given as:

$$\textbf{Adaptation}: \quad \frac{\{R\}\ \textbf{call}\,\omega(a):(e)\ \{S\}}{\{\exists k\ (R\ \wedge\ \forall a(S \Rightarrow T))\}\textbf{call}\,\omega(a):(e)\ \{T\}}$$

where k is the list of all variables free in R and S but not occurring in a, e or T. While the motivation for why this rule appears as it does is obscure at first, what we are saying, is that there must be some assignment to the free variables of R and S (i.e., k) which makes a true precondition R to the call of ω, such that for any possible alteration within ω to the parameter a, S will still imply T. If so, then we can use our "rule of consequence" above to produce T as a postcondition.

With these rules of inference, we have all the power to extend proofs to many realistically large and complex programs. We have added arrays and procedure calls to mimic the behavior of most programming language constructs.

3 Functional correctness

Let α be a string representing a source program. For example, a Pascal program is just the linear string:

$$PROGRAM\ main(input, output); \cdots END.$$

We express the mathematical function denoted by program α by a *box* notation. $\boxed{\alpha}$ represents the function that computes the same values as program α.

Although a function is the intuitive model of a specification, often we simply want one feasible solution out of many possibilities. For example, by choosing one optimal strategy among several equivalent ones (e.g., equivalent optimal moves in a game-playing program), we usually do not care which solution the program employs. Because of this, we only need to define a specification as a *relation*. If r is such a specification relation, it is equivalent to a program p by the following *correctness* theorem:

Theorem: Program p is correct with respect to specification relation r if and only if $domain(r \cap \boxed{p}) = domain(r)$.

In other words, if we take the subset of r by considering those pairs in r that are also in function $\boxed{p}$ (i.e., $r \cap \boxed{p}$) we have a function. If this function has the same domain as r, then $\boxed{p}$ includes a pair of values for each member of relation r and we get a feasible (or correct) implementation of the specification. In what follows, however, we will use the simpler case where we have chosen the more restricted specification function f instead of the more general relation r with the corresponding correctness theorem of $f \subset \boxed{p}$.

A program is a sequence of declarations followed by a sequence of statements. Each maps a set of values for every variable in the program into a new set of values. Using techniques from denotational semantics (which we will discuss in greater detail in the next section) we can define the meaning of such a program as follows:

If *id* is a set of variable names and *value* is a set of values, a *state* is a function with the signature $state : id \rightarrow value$. A *state* represents the formal model for program storage (e.g., activation records).

If *expr* is an expression, $\boxed{expr}$ is a function which maps a state into values, or $\boxed{expr} : state \rightarrow id$. For example, if (x, a) and (y, b) represent entries in the state function S representing variables x and y, then $\boxed{x+y}(S)$ is defined to be the function with $\boxed{x}(S) + \boxed{y}(S)$ as a value. If we define $\boxed{x}(S)$ to be $S(x)$, then $S(x) = a$ which agrees with our intuitive definition that $\boxed{x+y}(S) = a + b$.

If s is a statement, then $\boxed{s}$ is a function that maps a *state* into a *state*, i.e., each statement maps a set of values for all variables into a new set of values. If s is a declaration, then the resulting state includes a $(id, value)$ pair for the newly declared variable. For example, if s is the state $\{(x, 1)\ ,\ (y, 2)\}$, then the function $\boxed{y \leftarrow x}$ applied to s results in the state $\{(x, 1)\ ,\ (y, 1)\}$.

It is easy to see the correspondence between sequential execution and function composition. If s is the sequence $s_1; s_2; \ldots s_n$ of statements, then

$$\boxed{s} = \boxed{s_1; s_2; \ldots s_n} = \boxed{s_1} \circ \boxed{s_2} \circ \ldots \boxed{s_n} = s_n\ \ldots\ s_2(\ s_1()\ ) \ldots)$$

The function $\boxed{p}$ for program:

$$PROGRAM\ MAIN(INPUT, OUTPUT); BEGIN\ S1; S2; \cdots END.$$

is given by: $\boxed{PROGRAM\ MAIN(INPUT, OUTPUT)} \circ \boxed{S1} \circ \boxed{S2} \circ \ldots \boxed{\cdot}$ where the signature for $\boxed{PROGRAM \ldots}$ is $value \rightarrow state$, for $\boxed{\cdot}$ it is $state \rightarrow value$, and $state \rightarrow state$ for all other statements. Hence a program maps a value to a value and is composed of functions that map states to states. Details of how to handle individual statement types like assignments, conditionals and iteration are given later.

With this notation, there are three separate activities that can be investigated:

1. If f is a function and if p is a program, show $\boxed{p} = f$, i.e., verification.
2. If f is a function, develop program p such that $\boxed{p} = f$, i.e., program design. As a practical matter we only care that $f \subset \boxed{p}$ since any value in $\boxed{p}$ and not in f represents a value computed by the program which is outside of its specifications and not of interest to us.

3. If p is a program, then find a function f such that $\boxed{p} = f$; i.e., reverse engineering. Given a program, determine its specifications. Some heuristics are given, but the basic method is to "guess" a solution and show by rules (1) and (2) above that it is the correct solution.

3.1 Symbolic Execution

Symbolic execution is an aid in showing functional composition. In order to show that $\boxed{p} = f$, symbolically execute program p and show that the resulting function is the same as f. For example, consider the sequence:

$$x \leftarrow x + 1;$$
$$y \leftarrow x + y;$$
$$x \leftarrow y + 1$$

Since we know that $\boxed{x \leftarrow x + 1; y \leftarrow x + y; x \leftarrow y + 1} = \boxed{x \leftarrow x + 1} \circ \boxed{y \leftarrow x + y} \circ \boxed{x \leftarrow x + 1}$, we can symbolically execute each statement function. We use a trace table where we write under **Part** the relevant statement function and under each relevant variable the new value that results from that execution by substituting into the statement function the current value that each variable has at that point in the computation. This results in a new function that can transform each variable into its new value. For the above, we get the trace table:

Part	**x**	**y**
$x \leftarrow x + 1$	$x + 1$	
$y \leftarrow x + y$		$(x + 1) + y$
$x \leftarrow y + 1$	$(x + 1 + y) + 1$ = x+y+2	

This states that simultaneously x is transformed by the function $\boxed{x \leftarrow x + y + 2}$ and y is transformed by $\boxed{y \leftarrow x + y + 1}$.

The extension of the trace table to handle conditionals (e.g., **if** statements) requires the use of a **condition** column. We write the predicate that must be true at that point in the computation for that execution path to proceed and develop trace tables for each such path through the program.

For example the program sequence:

```
$x \leftarrow x + y$;
**if** $x > y$ **then**
    $x \leftarrow x - 1$
```

has two possible execution sequences (i.e., $x > y$ and $x \leq y$), and two corresponding traces:

Part	**Condition**	**x**	**y**
$x \leftarrow x + y$		$x + y$	
$if\ x > y$	$(x + y) > y$		
$x \leftarrow x - 1$		$(x + y) - 1$	

Part	**Condition**	**x**	**y**
$x \leftarrow x + y$		$x + y$	
$if\ x > y$	$(x + y) \leq y$		

We summarize these two tables by saying that the function they represent is: "if $x + y > y$ then the function is $\boxed{x \leftarrow x + y - 1}$ and if $x + y \ \leq y$ then the function is $\boxed{x \leftarrow x + y}$." In the next section we show how to write this as a conditional assignment function.

3.2 Design Rules

As given earlier, software development consists of (1) specification design, (2) formalizing the specification and (3) developing the source program. We use a functional notation for step (2) of the process that is closely tied to the eventual source program. This notation includes: (1) Concurrent assignment; (2) Conditional assignment and (3) Loop verification. This notation was strongly influenced by the earlier work by McCarthy LISP.

3.2.1 Design of Assignment Statements

Concurrent assignment is defined as simultaneous assignment. The function: $(x, y, z \leftarrow y, z, x)$ simultaneously accesses the current values of variables y, z and x and stores them, respectively into variables x, y, and z. Mathematically, we are stating that the state function that results will have the same values for all state variables other than x, y, and z, and that these three will have new values.

Given statement p, showing that $\boxed{p}$ does implement this concurrent assignment is simply a matter of building its trace table. The more interesting problem is how to develop p given some concurrent assignment as its specification. This leads to three design heuristics for concurrent assignment:

1. All values on the right side of the intended concurrent assignment (i.e., all values that are needed by a left hand side variable) must be computable at each step.
2. At each step, if a variable can be assigned its intended value, do so; otherwise introduce a temporary variable and assign to it a value that must be preserved.
3. Stop when all variables on the left side of the intended concurrent assignment have been assigned their intended values (i.e., finished).

If we "execute" a trace table as we develop each assignment statement, then we are also verifying that the design works as we wish. Once the values in the trace table are the values we desire, then we have shown that the assignment statements we have written do indeed implement the intended concurrent assignment.

An important point to remember is that the three design rules are heuristics and not an algorithm. They indicate how to search for a solution and how to check if the solution is correct. They do not give the solution. We have not replaced the art of programming by an implementable methodology to automatically build correct programs from specifications.

3.2.2 Design of Conditional Statements

The conditional assignment is the formal model of conditionals. If b_i is a boolean condition and c_i is a design function, then a conditional statement has the syntax: $(b_1 \rightarrow c_1)|(b_2 \rightarrow c_2)|\ldots|(b_n \rightarrow c_n)$ with the semantics of evaluating each b_i in turn, and setting the value of the conditional to be c_i for the first b_i which is true. If all b_i are false, then the statement is undefined. (This is similar to the **cond** of LISP.) If b_n is the default case (i.e., the expression **true**), then it can be omitted with the last term becoming (c_n). The *Identity* function is written as $()$.

Several theorems, which can all be verified by appropriate trace tables, involving conditional statements will be used in this paper. Their inference rules are given as follows:

$$\frac{(a \rightarrow b) \mid (\neg a \rightarrow c)}{(a \rightarrow b) \mid (c)}$$

$$\frac{(a \rightarrow (b \rightarrow c))}{(a \wedge b \rightarrow c)}$$

$$\frac{(a \rightarrow c) \mid (b \rightarrow c)}{(a \vee b \rightarrow c)}$$

$$\frac{(a \rightarrow (b \vee c))}{(a \rightarrow b) \mid (a \rightarrow c)}$$

The source program for this design is simply a series of **if** statements that test each condition in turn. For example, given the specification:

$$(b_1 \rightarrow c_1)|(b_2 \rightarrow c_2)|\ldots|(b_n \rightarrow c_n)$$

the program can be written directly as:

```
if $b_1$ then $c_1$
else if $b_2$ then $c_2$
else if $b_3$ then $c_3$
...
```

If all the b_i are false, since the code is everywhere defined, the specifications are actually a (correct) subset of this source program.

3.2.3 Verification of Assignment and Conditional Statements

Assume p is the program to be verified and it consists of only **if** and assignment statements. There are only a finite number of execution paths through the program. For each path, compute the condition that must be true to execute that path and use a trace table to determine what happens to the variables by executing that given path. Assume $p_1, p_2, \cdots$ are the conjunctions of all conditions on each execution path, and $a_1, a_2, \cdots$ are the corresponding concurrent assignments. The function f that this implements is:

$$(p_1 \rightarrow a_1)|(p_2 \rightarrow a_2)|\cdots|(p_n \rightarrow a_n)$$

Consider the following example:

```
$x \leftarrow x + y$            $\{1\}$
$y \leftarrow y - x$;           $\{2\}$
if $x + y > 0$ then
  $y \leftarrow x + y$          $\{3\}$
  else
  $y \leftarrow -x - y$         $\{4\}$
```

This has two execution sequences, 1-2-3 and 1-2-4, with two different traces:

if is true

Part	**Condition**	**x**	**y**
$x \leftarrow x + y$		$x + y$	y
$y \leftarrow y - x$		$x + y$	$-x$
$if\ x + y > 0$	$(x + y) - x > 0$		
$y \leftarrow x + y$			$x + y - x = y$

if is false – so ¬ (if) is true

Part	**Condition**	**x**	**y**
$x \leftarrow x + y$		$x + y$	y
$y \leftarrow y - x$		$x + y$	$-x$
$if\ x + y > 0$	$(x + y) - x \leq 0$		
$y \leftarrow -x - y$			$-(-x) - (x + y) = -y$

This gives the function: $(y > 0 \rightarrow x, y \leftarrow x + y, y)|(y \leq 0 \rightarrow x, y \leftarrow x + y, -y)$ or since the assignment to y (i.e., $(y > 0 \rightarrow y \leftarrow y)$ and $(y \leq 0 \rightarrow y \leftarrow -y)$) is just function $abs(y)$, the function reduces to: $(x, y \leftarrow x + y, abs(y))$.

While we could have left our answer as a conditional assignment, replacing it as a concurrent assignment using the absolute value function leads to a more understandable solution. Knowing when (and how) to apply such reductions is probably as complex an issue as encountered in axiomatic verification.

3.2.4 Verification of While Loops

In order to handle full program functionality, we must address loops. Given a functional description f and a **while** statement p, we first describe three verification rules that prove that f and p are equivalent. These will be denoted as **V.I-III**. Once we have these verification conditions, we would like to use them as design guidelines to help develop p, given only f. We call these five design rules **B.I-V**.

The **while** statement $\boxed{while\ b\ do\ d}$ is defined recursively via the **if** to mean:

$$\boxed{while\ b\ do\ d} = \boxed{if\ b\ then\ begin\ d; while\ b\ do\ d\ end}$$

That is, if b is true, perform d and repeat the **while**. Via a simple trace table we get the same result as:

$$\boxed{while\ b\ do\ d} = \boxed{if\ b\ then\ d; while\ b\ do\ d} = \qquad (**)$$
$$\boxed{if\ b\ then\ d} \circ \boxed{while\ b\ do\ d}$$

The three verification conditions are given as:

$$\begin{array}{ll} \mathbf{V.I} & f = \boxed{if\ b\ then\ d} \circ f \\ \mathbf{V.II} & domain(f) = domain(\boxed{while\ b\ do\ d}) \\ \mathbf{V.III} & (\neg b \rightarrow f) = (\neg b \rightarrow ()) \end{array}$$

We get **V.I** if we let f be the meaning of the **while** statement, i.e., $f = \boxed{while\ b\ do\ d}$, substituting back into (**), we get the first condition.

For **V.II**, what other conditions are on f insure that it is indeed the specification of the **while** statement? If f is undefined for some input x, then both sides of the equation are undefined. In order to insure that this cannot happen, we require that f is defined whenever $\boxed{while}$ is defined, or that $domain(\boxed{while}) \subset domain(f)$. (Note: For ease in reading, we will use $\boxed{while}$ to stand for $\boxed{while\ b\ do\ d}$.)

Similarly, $\boxed{while}$ is everywhere the identity function then any f will fulfill the equation, because, if $\boxed{while}$ is everywhere the identity, then similarly $\boxed{if\ b\ then\ d}$ must also be the identity and the recursive equation reduces to $f = () \circ f = f$. Thus we must also have $domain(f) \subset domain(\boxed{while})$.

For **V.III**, consider any state $s \in domain(\boxed{while})$. If $\boxed{b}(s)$ is true, i.e., expression b in state s is true, then from (**), $s_1 = \boxed{d}(s)$ and $s_1 \in domain(\boxed{while})$. This will be true, for $s_2, s_3, \cdots$ until at some point, $\boxed{b}(s_n)$ is false and both $\boxed{if\ b\ then\ d}(s_n)$ and $\boxed{while\ b\ do\ d}(s_n)$ equal s_n.

This s_n is a member of $domain(\boxed{while})$ and of $range(\boxed{while})$. More importantly, if $\boxed{b}(s)$ evaluates to false, then $\boxed{while}(s) = s$, or stated another way: $\boxed{while}(s) = s$ for all states s where $\boxed{b}(s)$ is false. This is just a restriction on the $\boxed{while}$ function to those states where b is false, which is the function $(\neg(b) \rightarrow \boxed{while})$. This must be equal to the identity function () also restricted to the same domain, or just: $(\neg(b) \rightarrow ()\ )$. Any candidate function f must also have this property.

3.2.5 Design of While Loops

Consider the problem of designing a loop. Given a specification f, how can we design a **while** from the three statement verification conditions given above?

From V.III, the **while** terminates when $\boxed{b}$ evaluates to false, and $range(\boxed{while})$ is just the set of states where $\boxed{b}$ is false. But since we can apply $\boxed{while}$ to this state initially, it is also part of $domain(\boxed{while})$. Therefore, $range(\boxed{while}) \subset domain(\boxed{while})$ (see Figure 5). Since f must also have this property, we get:

D.I $range(f) \subset domain(f)$.

Similarly, we have shown that for an f where $\boxed{b}(s)$ is false, $\boxed{while}(s) = s$, we must also have $f(s) = s$,

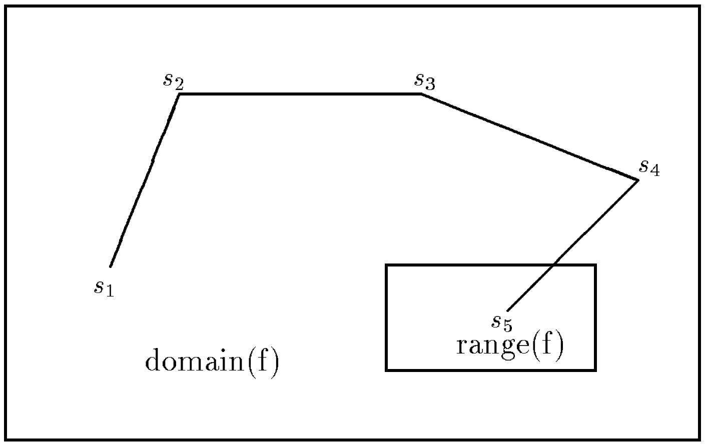

Figure 5: Domain and range of while function

because if $\boxed{\text{b}}$(s) is false, the body d is not executed. But these are just the points in range(f). Therefore, we get the second design constraint:

D.II *if* $s \in range(f), then\ f(s) = s$.

D.I and D.II must be true if f is the meaning of a **while** statement. Therefore they show the existence of a possible solution. From D.II, we know f must be an identity on $range(f)$ in order to be implemented with a **while**. We can restate this as:

D.III $\boxed{b}$ evaluates to true in $domain(f) - range(f)$ and false in $range(f)$.

Similarly to the assignment design, we develop the **while** loop via:

D.IV Develop d so that all values are preserved for f.

D.V Show that f is everywhere defined, i.e., the loop must terminate for all $x \in domain(f)$.

3.3 Multiplication revisited

In order to demonstrate this approach, let us revisit the multiplication program previously proven in Section 2.1.2:

```
        {$B \geq 0$}
1.      $MULT(A, B) \equiv$
2.          $a \leftarrow A$
3.          $b \leftarrow B$
4.          $y \leftarrow 0$
5.          **while** $b > 0$ **do**
6.              **begin**
7.              $y \leftarrow y + a$
8.              $b \leftarrow b - 1$
9.              **end**
        {$y = AB$}
```

Because of composition of assignment statement functions, it should be clear that we need to show:

$$\boxed{MULT(A,B)} = \boxed{a \leftarrow A} \circ \boxed{b \leftarrow B} \circ \boxed{y \leftarrow 0} \circ \boxed{while} = AB$$

While verification. The while loop adds ab to y and sets b to 0, therefore, we can use as a candidate function f for this loop the equation:

$$f(a,b,y) = (b > 0 \rightarrow (a, 0, y + ab)|()$$

To show this, we must show that the three properties **V.I–V.III** are true.

$f = \boxed{if\ b\ then\ d} \circ f$. In this case, predicate b is $b > 0$ and d is assignment statements 7 and 8 in the program.

This requires 4 trace tables since we have two choices for the predicate in $\boxed{if}$ and two choices for the predicate in f.

1. f_1 = both are true.

Part	**Condition**	**a**	**b**	**y**
if	$b > 0$			
$b \leftarrow b - 1$			$b - 1$	
$y \leftarrow y + a$				$y + a$
f	$b - 1 \geq 0$	a	0	$y + a + a(b-1) =$ $y + ab$

$$f_1(a,b,y) = \quad (b > 0 \wedge b - 1 \geq 0 \rightarrow (a, 0, y + ab)) = (b > 0 \rightarrow (a, 0, y + ab))$$

2. f_2 = if is false, f is true.

Part	**Condition**	**a**	**b**	**y**
if	$b \leq 0$			
f	$b \geq 0$	a	0	$y + ab$

$$f_2(a,b,y) = \quad (b \leq 0 \wedge b \geq 0 \rightarrow (a, 0, y + ab)) = (b = 0) \rightarrow (a, 0, y + ab)$$

3. $f_3 =$ *if is true, f is false.*

Part	**Condition**	**a**	**b**	**y**
if	$b > 0$			
$b \leftarrow b-1$			$b-1$	
$y \leftarrow y+a$				$y+a$
f	$b-1<0$			

$$f_3(a,b,y) = \quad (b > 0 \wedge b-1 < 0 \rightarrow (a, b-1, y+a)) = (b > 0 \wedge b < 1 \rightarrow (a, b-1, y+a))$$

In this case, there is no solution for $(b > 0 \wedge b < 1)$.

4. $f_4 =$ *both are false.*

Part	**Condition**	**a**	**b**	**y**
if	$b \leq 0$			
f	$b < 0$			

$$f_4(a,b,y) = \quad (b \leq 0 \wedge b < 0 \rightarrow ()) = (b < 0 \rightarrow ())$$

We then compute the required functionality for the **while** statement:

$$\begin{aligned} f = \quad & f_1|f_2|f_4 = \ (b \geq 0 \rightarrow (a, 0, y+ab))|(b < 0 \rightarrow ()) = \\ & (b \geq 0 \rightarrow (a, 0, y+ab))|() \end{aligned}$$

Now that we have defined f, we have to show that it meets the other two properties:

Property V.II *domain*$(f) =$ *domain*$(\boxed{\text{while}})$

Both functions are defined for all b, so domains are equal.

Property V.III $\neg b \rightarrow f = \neg \rightarrow ()$.

$$\begin{aligned} & \neg b \rightarrow f = \\ & b < 0 \wedge (b \geq 0 \rightarrow (a, 0, y+ab)|()) = \\ & b = 0 \rightarrow (a, 0, y+ab) = \\ & (a, 0, y) = () \end{aligned}$$

Program verification. Now that we have identified the function of the **while** statement, we can complete the proof of $MULT$ by a trace table of the four program components:

Part	**Condition**	**a**	**b**	**y**
$a \leftarrow A$		A		
$b \leftarrow B$			B	
$y \leftarrow 0$				0
f		a	0	$0 + AB = AB$

This clearly shows that $MULT(A, B) = AB$. We also need to show termination of the program, but the same argument used previously is still true here.

3.4 Data Abstraction and Representation Functions

The discussion so far has concentrated on the process of developing a correct procedure from a formal specification. However, program design also requires appropriate handling of data.

3.4.1 Data Abstractions

A *data abstraction* is a class of objects and a set of operators that access and modify objects in that class. Such objects are usually defined via the *type* mechanism of a given programming language, and a module is created consisting of such a type definition and its associated procedures.

Crucial to the data abstraction model is the isolation of the type definition and invocations of the procedures that operate on such objects. Each procedure has a well-defined input-output definition. The implementor is free to modify any procedure within a module as long as its input-output functional behavior is preserved, and any use of such a procedure can only assume its functional specification. The result of this is that rather than viewing a program as a complex interaction among many objects and procedures, a program can be viewed as the interaction among a small set of data abstractions–each relatively small and well defined.

Languages such as Ada (or C++) allows data abstractions to be built relatively easily since the object type can be specified as the **private** part of a **package** (or **class**) specification. Only the body of the **package** has access to the type structure while other modules have access only to the names of the procedures that are contained in the module.

However, even in older languages, such as C or Pascal, data abstractions form a good model of program design. Even though not automatically supported by the language, with reasonable care, programs can be designed which adhere to the data abstraction guidelines.

3.4.2 Representation Functions

A procedure within a data abstraction translates a high-level description of a process into a lower level programming language implementation. For example, suppose character strings up to some predefined maximum value are needed. Pascal, for example, only defines fixed length strings; therefore, we must implement this as objects using primitive Pascal data types.

In procedures outside of the defining module, we would like to refer to these objects (e.g., call them **Vstrings**) and be able to operate on them, while inside the module we need to operate on their Pascal representation (arrays of characters). In the former case, we call such functions **abstract functions** that define the functional behavior of the operation, while we call the latter case **concrete functions** that gives the details of the implementation.

For the **Vstring** example, we could define such a string via an abstract comment containing the functional definition:

$$\{abs : x_{abs} \ = < x_1, x_2, \ldots x_n >\}$$

while the concrete representation of a **Vstring** could be:

$$\begin{aligned}&con : x_{con} : record\\ &\quad chars : array(1 \ldots maxval)\ of\ char;\\ &\quad size : 0..maxval\\ &\quad end;\end{aligned}$$

To show that both representations are the same, we define a *representation function* which maps concrete objects into abstract objects by mapping a state into a similar state leaving all data unchanged except for those specific objects. Let r map a concrete object into its abstract representation. If **Cstrings** is the set of concrete strings (i.e., the set of variables defined by the above record description) and if **Vstrings** is the set of abstract strings, then we define a representation function r with the signature: $r : state \rightarrow state$ such that

$$r \ = \ \{(u, v) | u = v \ except\ that\ if\ u(x) \in Cstrings, then\ v(x) \in Vstrings\ \}$$

We simply mean that u and v represent the same set of variables in the program store except that each occurence of a concrete variable in u is replaced by its abstract definition in v.

For each implementation of a string we have its abstract meaning given by function r:

$$x_{abs} \leftarrow < x_{con}.chars[i] | 1 \leq i \leq x_{con}.size >$$

The purpose of a procedure in an abstraction module is to implement an abstract function on this abstract data. For example, if we would like to implement an *Append* operation, we can define $x \leftarrow Append(x, y)$ as:

$$\{abs : x_1, \ldots, x_n, \ldots, x_{n+m} \leftarrow x_1, \ldots x_n, y_1, \ldots, y_m\}$$

Similarly, we can define a concrete implementation of this same function as:

$$\begin{aligned}\{con : x.chars[n+1], \ldots, x.chars[n+m], x.size \leftarrow\\ y.chars[1], \ldots y.chars[y.size], x.size + y.size\}\end{aligned}$$

If x_{con} and y_{con} represent the concrete implementations of **Vstrings** x and y, and if x_{abbs} and y_{abs} represent their abstract representation, and if $Append_{con}$ and $Append_{abs}$ represent the concrete and abstract functions, we have:

$$\begin{aligned}x'_{con} &\leftarrow Append_{con}(x_{con}, y_{con})\\ x'_{abs} &\leftarrow Append_{abs}(x_{abs}, y_{abs})\end{aligned}$$

We want to know if both the concrete and abstract functions achieve the same result, or is the abstract representation of what we get by implementing $Append_{con}$ the same as our abstract definition of *Append*? This is just the result: Is $r(x'_{con}) = x'_{abs}$? We say that the representation diagram of Figure 6 *commutes* (i.e., either path from (x_{con}, y_{con}) to x'_{abs} gives the same result). We have to show that r applied to x'_{con} gives us x_{abs} (e.g., $x_1, x_2, \ldots, x_n, y_1, \ldots, y_m$).

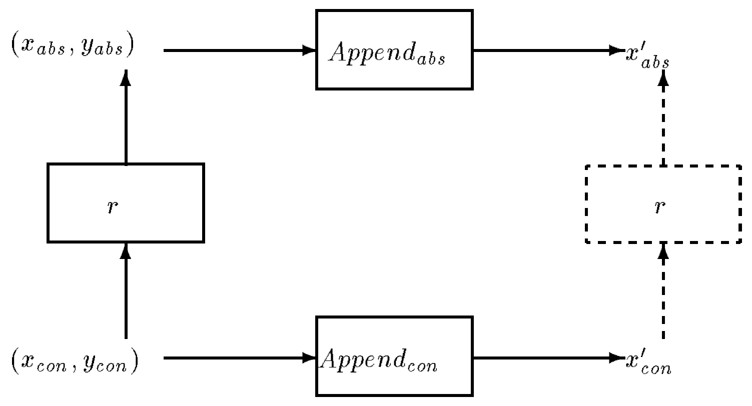

Figure 6: Commuting Representation Diagram

As given by our earlier correctness theorem, a program (e.g., $Append_{con}$) will often compute a value in a domain larger than necessary (e.g., $domain(Append_{abs}(x))$. Thus we actually want to show:

$$r \circ Append_{abs} \subset Append_{con} \circ r$$

3.5 A verification methodology

We have seemingly developed two mechanisms for designing programs: (1) a functional model for showing the equivalence of a design and its implementation, and (2) a commuting diagram for showing correct data abstractions. However, both are complementary ideas of the same theory. For example, the concrete design comment for *Append* in the previous section is just a concurrent assignment which we can translate into a source program via the techniques described earlier.

This leads to a strategy for developing correct programs:

1. From the requirements of a program, develop the abstract data objects that are needed.
2. For each object, develop abstract functions that may be necessary to operate on the abstract object.
3. Using the abstract object and operations as a goal, design the concrete representation of the object and corresponding representation function.
4. Design a concrete function for each corresponding abstract function.
5. Show that the representation diagram commutes. That is, the concrete function does indeed implement the abstract function.
6. Develop correct programs from each concrete function.

The importance of this technique is that it can be applied at any level of detail. In this paper, we obviously considered only short program segments. For larger programs, only those concepts that are critical to the success of a program need be formalized, although a long range goal would be to develop this or other techniques which can be applied to very large systems in their entirety. Its major difference from other verification techniques is that it forces the programmer or designer to consider the functionality of the program as a whole, and requires the designer to design data structures with operations that operate on those structures. Since this is central to the data abstraction model of program design, this technique is quite applicable to current thinking about programming.

4 Denotational semantics

In the previous section, the functional correctness model was presented. Here we look at statement functionality in greater detail.

This model is based upon a λ-calculus-like notation. We first summarize the λ-calculus and then extend it to programming language semantics.

4.1 The Lambda Calculus

The λ-calculus is a formal functional model used by Church to develop a theory of numbers. It is relevant to programming language design since it represents a typeless (hence simpler) model in the theory of denotational semantics.

λ-expressions are defined recursively as:

1. If x is a variable name, then x is a λ-expression.
2. If M is a λ-expression, then $\lambda x M$ is a λ-expression.
3. If F and A are λ-expressions, then (FA) is a λ-expression. F is the *operator* and A is the *operand.*

The following are all λ-expressions:

$$x \quad \lambda xx \quad \lambda xy \quad \lambda x(xy) \quad (\lambda x(xx)\lambda x(xx)) \quad \lambda x\lambda yx$$

Variables may be bound or free. In λxM, x is the *binding* variable and occurances of x in M are *bound.* A variable is *free* if it is not bound.

Any bound variable may have its name changed. Thus the λ-expression λxx is equivalent to λyy. $\lambda x\lambda xx$ is equivalent to $\lambda x\lambda yy$ since the variable x is not free in the original λ-expression λxx.

Informally, bound variables are "parameters" to the function described by the λ-expression, free variables are global. There is no concept of local variable.

This analogy shows that the λ-expression is a simple approximation to the procedure or subroutine concept in most algorithmic programming languages like Pascal, C, Ada or Fortran. λ-expressions are almost directly representable in LISP and λ-substitutions are a direct model of Algol-like procedures.

λ-expressions have only one operation – *reduction.* If (FA) is a λ-expression, and $F = \lambda xM$, then A may be substituted for all free occurrences of x in M. This is written as: $(\lambda xMA) \rightarrow F'$.

Examples:

$$\begin{array}{lll}
(\lambda x x y) & \rightarrow y & \\
(\lambda x(xy)y) & \rightarrow (yy) & \\
(\lambda x(xy)\,\lambda xx) & \rightarrow (\lambda x x y) & \rightarrow y \\
(\lambda x(xx)\,\lambda x(xx)) & \rightarrow (\lambda x(xx)\,\lambda x(xx)) & \rightarrow \ldots
\end{array}$$

Note that the third expression does not terminate. If we have $F = (\lambda x M\ A)$, substitution of A for x in M results in F again. This is a non-terminating reduction. This leads us to:

Church-Rosser property. If two different reductions of a λ-expression terminate, then they are members of the same value class.

The λ-calculus was originally developed as a logical model of the computational process. We can use such expressions to model our understanding of arithmetic. The following is a very brief summary of this use.

Boolean values as λ-expressions

Objects will be modeled by functions.

True (T) will be defined as: $\lambda x \lambda y x$. (Of a pair, choose the first.)

False (F) will be defined as: $\lambda x \lambda y y$. (Of a pair, choose the second.)

We have defined these objects so the following properties are true:

$$((T\ P)Q) \rightarrow P$$
$$((F\ P)Q) \rightarrow Q$$

Proof for T: $((T\ P)Q) \rightarrow ((\lambda x \lambda y x\ P)Q) \rightarrow (\lambda y P\ Q) \rightarrow P$

Proof for F: $((F\ P)Q) \rightarrow ((\lambda x \lambda y y\ P)Q) \rightarrow (\lambda y y\ Q) \rightarrow Q$

Given these constants T and F, we can define the boolean functions:

$$\begin{aligned}
\mathit{not} &\equiv \lambda x((xF)T) \\
\mathit{and} &\equiv \lambda x \lambda y((xy)F) \\
\mathit{or} &\equiv \lambda x \lambda y((xT)y)
\end{aligned}$$

These strange expressions have all the properties we need, e.g.

$$(\mathit{not}\ \ T) = (\lambda x((x\ F)T)T) \rightarrow ((T\ F)T) \rightarrow F$$
$$(\mathit{not}\ \ F) = (\lambda x((x\ F)T)F) \rightarrow ((F\ F)T) \rightarrow T$$

From these logical definitions, we can build up the entire theory of integers, and from that, the recursively enumerable sets. However, in this paper, we are more interested in the development of programming language constructs.

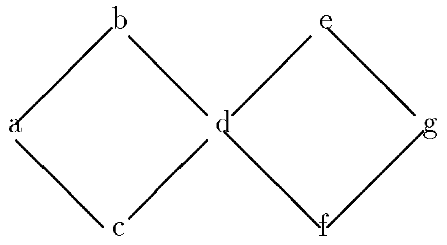

Figure 7: Lattice

4.2 Lattice theory of datatypes

λ-expressions either reduce to constants (*value*) or to other λ-expressions. Thus all λ-expressions are solutions to the functional equation:

$$value = constant + (value \rightarrow value)$$

But there exists no set solving this since the cardinality of functions from $(value \rightarrow value)$ is greater than the cardinality of *value*s.

Assume we have infinite objects with some finite approximation as a representation. Object a approximates object b (written a *ap* b) if b is more accurate than a.

Examples:

(a) a *ap* b if whenever a is defined, b is defined and gives the same answer.

(b) Let $[x_1, x_2]$ be a line interval. $[x_1, x_2]$ *ap* $[y_1, y_2]$ if $x_1 \leq y_1 \leq y_2 \leq x_2$. That is $[y_1, y_2]$ is contained in $[x_1, x_2]$.

We are interested in the least upper bound of approximations. $F = \{f_1, \ldots, f_n\}$ where

$$f_1 \; ap \; f_2 \; ap \; \ldots \; ap \; f_n$$

Since we require that such limits exist for every subset of a datatype, we have:

Axiom 1: A *datatype* is a complete lattice under partial ordering *ap*. Objects of this lattice are called *domains*. That is, for two objects a and b, a *ap* b or b *ap* a or $a \neg ap\, b$. Also, if a *ap* b and b *ap* c then a *ap* c.

Examples:

(a) Consider the lattice with 7 objects as given in Figure 7. In this case, i *ap* j means that i is "lower" than j. We then have:

$a\; ap\; b$	$d\; ap\; b$	$d\; ap\; e$	$g\; ap\; e$
$c\; ap\; a$	$c\; ap\; d$	$f\; ap\; d$	$f\; ap\; g$

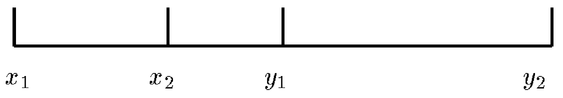

Figure 8: Line interval

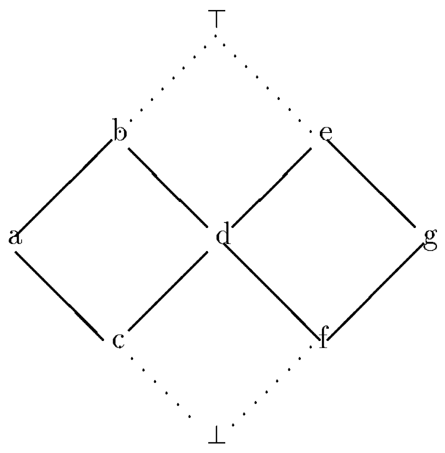

Figure 9: Complete lattice

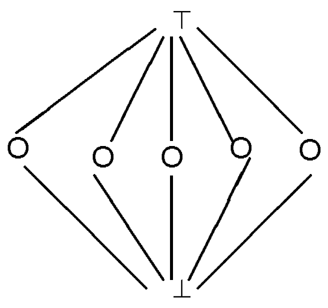

Figure 10: Primitive domain

(b) Consider $[x_1, x_2]$ and $[y_1, y_2]$ on the line interval (Figure 8). For set $\{[x_1, x_2], [y_1, y_2]\}$, there is no *lub*. But one must exist by Axiom 1'. So we hypothesize the existence of $\top$ (pronounced "top") and $\bot$ (pronounced "bottom"). This gives the following relationships for our previous lattice, as given in Figure 9.

(c) For discrete sets, we call Figure 10 a *primitive domain.*

(d) The simpliest primitive domain will consist only of $\top$ and $\bot$ and we call this the *simple* domain (S).

4.2.1 Continuous functions

Axiom 2: Mappings between domains are monotonic. That is, if $f : D \to D'$, then

$$if\ x\ ap\ y\ then\ f(x)\ ap'\ f(y)$$

A set $X \subset D$ is *well defined* iff $\exists p$ and

$$p : D \to \mathcal{S} \text{ such that } X = \{w | w \in D\, and\, p(w) = \top\}.$$

Axiom 3: Mappings under domains are continuous.

This means: $f(lub(X)) = lub\{f(x)|x \in X\}$. Continuous functions between domains also form a domain: $\mathcal{S} \to \mathcal{S}$. We get the lattice:

$$\begin{tikzcd}
(\mathcal{S}\, x)\mathcal{S} : \top \arrow[d, dash] \\
(\mathcal{S}\, x)\mathcal{S} : x \arrow[d, dash] \\
(\mathcal{S}\, x)\mathcal{S} : \bot
\end{tikzcd}$$

4.2.2 Recursive functions

Extend integers to a complete lattice:

n_1	n_2	$n_1 \times n_2$
i_1	i_2	$i_1 \times i_2$
$\bot$	i	$\bot$
$\top$	i	$\top$
i	$\bot$	$\bot$
i	$\top$	$\top$
$\top$	$\top$	$\top$
$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$

As an example to analyze, lets look at the factorial function. We will define it as follows:

$FACT \equiv (num\, k)num : if\ k = 0\, then\, 1\, else\ k \times FACT(k-1)$.

This equation can be rewritten as: Find the function $FACT$ that solves the equation: $FACT = F(FACT)$, where F is a function from $(num \to num)$ to $(num \to num)$, i.e., given a function f from $(num \to num)$ as an argument, $F(f)$ is a function from $(num \to num)$.

We can then define F as:

$$\begin{aligned} F \equiv \quad & ((num \rightarrow num)FACT)(num \rightarrow num) : \\ & \quad (num\ k)num : if\ k = 0\ then\ 1\ else\ k \times FACT(k-1) \end{aligned}$$

This might be clearer (or totally obscure!) by saying that the $FACT$ function is a solution to $x = F(x)$ for F defined as:

$$\begin{aligned} F \equiv \quad & ((num \rightarrow num)x)(num \rightarrow num) : (num\ k)num : \\ & \quad if\ k = 0\ then\ 1\ else\ k \times x(k-1) \end{aligned}$$

F takes argument $(num \rightarrow num)$ (i.e., a function) and produces a $(num \rightarrow num)$ (i.e., another function) such that the function argument takes a num parameters and applies the transformation $(k = 0 \ \rightarrow 1)|(k \times x(k-1))$. We want the function x such that $x = F(x)$, then this x is called a *fixed point* of F. Consider $f : D \rightarrow D$. x is a *fixed point* of f iff $x = f(x)$. A fixed point is a *least fixed point* if for all fixed points $x_1, x_2, \ldots x_n$, $x\ ap\ x_1$, $x\ ap\ x_2$, $\ldots$, $x\ ap\ x_n$.

A recursive function has many fixed points. For example, for some n let:

$$g = (num\ x)num : if\ x = \perp\ then\ \perp\ else\ if\ x = \top\ then\ \top\ else\ n$$

For any x, you get $x = g(x)$, so all x are fixed points of g.

It can be shown that any continuous function on a complete lattice has a least fixed point. Moreover, if $f : D \rightarrow D$ is continuous, the least fixed point $x = f(x)$ is given by:

$$x = lub\{f_n(\perp) | n = 0, \ldots k\}$$

where

$$f_n(\perp) = f(f \ldots f(\perp) \ldots)$$

and

$$f_0(\perp) \equiv \perp$$

Evaluation of FACT function

So where are we? Consider the function $FACT$ defined previously. The least fixed point is derived from the previous equation to be:

$$F_i = (num\ k)num : if\ k = 0\, then\, 1\, else\ k \times F_{i-1}(k-1)$$

with $F_0 = \perp$.

$i = 0$	$F_0 = \perp$ *for all* k
$i = 1$	$F_1 = if\ k = 0\ then\ 1\ else\ k \times F_0(k-1)$
	$k = \perp\ F_1(k) = \perp$
	$k = 0\ \ F_1(k) = 1$
	$k = 1\ \ F_1(k) = 1 \times F_0(0) = \perp$
	$k \geq 2\ \ F_1(k) = \perp$
$i = 2$	$F_2 = if\ k = 0\ then\ 1\ else\ k \times F_1(k-1)$
	$k = \perp\ F_2(k) = \perp$
	$k = 0\ \ F_2(k) = 1$
	$k = 1\ \ F_2(k) = 1 \times F_1(0) = 1$
	$k = 2\ \ F_2(k) = 1 \times F_1(1) = \perp$
...	...

Each F_i defines $FACT$ for arguments $0 \leq k < i$, and for all $k \geq i$, we have $F_i(k) = \perp$.

4.3 Programming language semantics

The use of domains and fixed points allows us to describe programming language semantics. The model is a form of operational semantics, since we are "tracing" through the execution of each statement type in order to determine its effect on a higher level "interpreter."

However, unlike traditional interpreters that you may be familiar with, we will not consider a "machine" that symbolically executes the program. In this case we will consider a program to be a function, much like the functional correctness model considered earlier. Each statement of the language will be a function, so like the earlier model, we will model the execution of successive statement by determining what the composition function of the two must be.

In the earlier model, we would compute the composition function and gave techniques to determine if this composition was the same as an externally given specification of these same statements. In this present chapter, we will simply determine what the composition function must be by describing the transformations that occur to the functions, themselves.

Consider the language we have been using in this paper. We can write down a domain equation from each of these BNF rules:

$$\begin{array}{lll} Stmt = & (Id \times Exp) & \text{Domain of assignments} \\ & +(Stmt \times Stmt) & \text{Domain of sequences} \\ & +(Exp \times Stmt \times Stmt) & \text{Domain of conditionals} \\ & +(Exp \times Stmt) & \text{Domain of iterations} \end{array}$$

Since **begin** $< stmt >$ **end** only groups statements and does not alter the semantics of the internal $< stmt >$, it was ignored by the domain equation above.

4.3.1 The simple model

As with any interpreter, we need to understand the underlying "machine." We will assume that each identifier refers to a specific location in memory. Aliasing and parameter passing are not allowed, and there is no such concept as a pointer variable. We will extend this model in the next section to include these features.

Therefore for each identifier, there is a unique location in memory containing its value. Stated another way, we can model our concept of memory as a function that returns for each identifier, its unique value. We will call such a function a *store*.

We first must identify three important domains:

Values of storable values. These are the results of computations and in our example will be the results of expression evaluation. These are the values that identifiers can take.

Eval of expression values. These are the results of any computations but may not necessarily be storable into an identifier. In our example, we will assume that these can be boolean or number. Number will be modeled by the primitive domain *num* given earlier and boolean will be the domain *bool* given by:

$$\bot, false, true, \top$$

Denotable values. These include objects that can be named by identifiers (e.g., procedures, labels, arrays). We will not be using these initially, but the next section introduces this concept.

The storage for a program is a function that maps each location (or *id*) into its appropriate value. The effects of executing a statement is to create a slightly modified storage. Thus we have a new function (similar to the previous one) that now maps each *id* into a value – possibly the same or different one. Thus we are viewing execution as the composition of the value storage function followed by the function that describes the execution of a particular statement.

We view a *store* as a mapping from identifiers to storable values, or a domain of type $id \rightarrow value$, where *id* is the primitive domain of identifiers in the language. We will call this the *program state*.

In defining the semantics for a programming language, there are three basic functions we need to represent:

1. We need to describe the semantics of a *program*. In this case, the syntax of a program defines a program which has the effect of mapping a number to another number. Or in other words, we are looking for a function $\mathcal{M}$ with signature:
$$\mathcal{M} : prog \rightarrow [num \rightarrow num]$$
2. We need to describe a statement in the language. Each statement maps a given state to another state, or:
$$\mathcal{C} : stmt \rightarrow [state \rightarrow state]$$
This simply states that each syntactic statement (domain of $\mathcal{C}$) is a function from *state* to *state*. So each unique statement will take a program state (or mapping from identifier to value) and produce a new mapping from identifier to value that is the result of all previous statements including this new statement.
3. The statement function $\mathcal{C}$ depends upon the value of various expressions, so we need to understand the effects of expression evaluation in our language. Each expression is a syntactic entity involving identifiers and accesses the storage to produce a value. Thus from the set of syntactic expressions, a set of storable values produces an expression value. This gives the signature for expression evaluation as:
$$\mathcal{E} : exp \rightarrow [state \rightarrow eval]$$
As a simple example, if *exp* is the expression $a+b$, then the evaluation function is: $\mathcal{E}(a+b) : state \rightarrow eval$ and applying this function to the state s gives us the function:

$$\mathcal{E}(a+b)(s) \;=\; a(s)+b(s)$$

We will usually write $\mathcal{E}(a+b)$ as $\mathcal{E}\{\!\{a+b\}\!\}$ by putting syntactic arguments in set braces rather than parentheses. It should also be clear, that this notation is the same as $\boxed{a+b}$ when we described functional correctness earlier.

We can now model the domains for our programming language:

$state \equiv id \rightarrow value$	*program states*
id	*identifiers*
$value \equiv eval$	*values*
$eval \equiv num + bool$	*expression values*
num	*integers*
$bool$	*booleans*
exp	*expressions*
$stmt$	*statements*

To define our language, we need to define a function of type *state* $\rightarrow$ *state* for each of the syntactic statement types. For ease in readability we will use a λ-like notation for certain constructs. The term: *let* $x \leftarrow a$ *in body* will be used to mean $((x) : body)(a)$ and has a meaning similar to the λ-expression: $(\lambda x\ body\ a)$.

Recursive definitions will be denoted as: $f = rec\ F(f)$ and will denote the least fixed point of F.

Some auxilliary functions we will use include:

- The conditional $b \rightarrow v_1, v_2$ is defined as follows:

b	$b \rightarrow v_1, v_2$
$\top$	$\top$
$true$	v_1
$false$	v_2
$\bot$	$\bot$

- If x is of domaintype $D \rightarrow D'$, then the expression $x[v/e]$ is defined as:

$$x[v/e] \equiv \quad (D\ d)D' : if\ d = v\ then\ e\ else\ x(d)$$

This has the intuitive meaning of changing the v component of x to e and represents the basic model of an "assignment."

- If y is a domain of type $y = \ldots + x + \ldots$ then the projection is defined as: For any $y \in Y$ the projection of y into Y, written $y|Y$ is defined as:

$$y|X = y\ for\ y \in X,\ or\ \bot\ if\ \neg\ y \in X$$

Statement semantics

begin No real semantics added by the **begin** ...**end** sequence.

$$\mathcal{C}\{\!\{begin\ stmt\ end\}\!\} \;=\; \mathcal{C}\{\!\{stmt\}\!\}$$

composition Result is transform by the first statement first, then the second.

$$\mathcal{C}\{\!\{stmt_1\ ;\ stmt_2\}\!\} \;=\; (state\ s)state : \mathcal{C}\{\!\{stmt_2\}\!\}(\mathcal{C}\{\!\{stmt_1\}\!\}(s))$$

assignment In this case create a new storage map resulting from the new denotable value. Note that this is just the array assignment axiom we already discussed in § 2.2.

$$\begin{aligned}\mathcal{C}\{\!\{id \leftarrow exp\}\!\} \;=\;& (state\ s)state : ((value\ v)state : s[id/v])\\ & ((\mathcal{E}\{\!\{exp\}\!\}(s)) \mid value)\end{aligned}$$

if This simply determines which of two functions to evaluate by first evaluating $\mathcal{E}$ on the expression, applying this to the boolean function, which then evaluates $stmt_1$ or $stmt_2$ as appropriate.

$$\begin{aligned}\mathcal{C}\{\!\{if\ exp\ then\ stmt_1\ else\ stmt_2\}\!\} \;=\;& (state\ s)state : ((bool\ b)state \rightarrow state :\\ & (b \rightarrow \mathcal{C}\{\!\{stmt_1\}\!\})\\ & |(\mathcal{C}\{\!\{stmt_2\}\!\}))\\ & ((\mathcal{E}\{\!\{exp\}\!\}(s)) \mid bool)(s)\end{aligned}$$

while This recursive definition is similar to the functional correctness model of § 3.

$$\begin{aligned}\mathcal{C}\{\!\{while\ exp\ do\ stmt\}\!\} =\;& rec(state\ s)state : ((bool\ b)state \rightarrow state :\\ & (b \rightarrow \mathcal{C}\{\!\{stmt\}\!\} \circ \mathcal{C}\{\!\{while\ exp\ do\ stmt\}\!\})|((state\ s')state : s'))\\ & \quad ((\mathcal{E}\{\!\{exp\}\!\}(s)) \mid bool)(s)\end{aligned}$$

These definitions correspond to the intuitive meanings we usually associate with these statements.

4.3.2 Pointers and Aliasing

Our initial model assumed that every identifier referenced a unique value. However, in almost every language, the notion of a reference exists in some form. Even if no explicit pointer exists, the notion is used to describe mechanisms like parameter passing to procedures. This also allows for *aliasing* or the ability of permitting multiple identifiers address the same location in memory.

In order to handle this more explicitly, the notion of a "program store" must be extended to the domain of denotable values. The domain *dval* of denotable values includes the primitive domain *loc* of locations. We add the concept of an *environment* as a mapping from identifiers to these denotable values, or the domain: $env = id \rightarrow dval$. The "program state" now becomes a "machine store," or a member of the domain: $store = loc \rightarrow value$.

If E is an environment, then given any identifier id, $E(id)$ becomes a function $loc \rightarrow dval$. For languages without pointers, we can assume that *loc* will be a constant, so we get the set of environments: $E : id \rightarrow dval$. However, the addition of *loc* allows us to define identifiers whose location in our store changes.

We need to redefine our semantics to determine the effects of each statement relative to an environment. The previous $\mathcal{C} : stmt \rightarrow [state \rightarrow state]$ now becomes $\mathcal{C}' : stmt \rightarrow [env \rightarrow [store \rightarrow store]]$. The expression evaluation function $\mathcal{E}$ now becomes $\mathcal{E}'$ to interpret the meanings of expressions relative to both an environment and a store.

The modified semantics become:

$$\mathcal{C}'\{\!\!\{begin\ stmt\ end\}\!\!\} = \mathcal{C}'\{\!\!\{stmt\}\!\!\}$$

$$\mathcal{C}'\{\!\!\{stmt_1\ ;\ stmt_2\}\!\!\} = (env\ e; store\ s)store : \mathcal{C}'\{\!\!\{stmt_2\}\!\!\}(e)(\mathcal{C}'\{\!\!\{stmt_1\}\!\!\}(e)(s))$$

$$\begin{aligned}\mathcal{C}'\{\!\!\{id \leftarrow exp\}\!\!\} = {} & (env\ e; store\ s)store : ((loc\ a; value\ v)store : s[a/v]) \\ & \quad (e(id))((\mathcal{E}'\{\!\!\{exp\}\!\!\}(e)(s)) \mid value)\end{aligned}$$

$$\begin{aligned}\mathcal{C}'\{\!\!\{if\ exp\ then\ stmt_1\ else\ stmt_2\}\!\!\} = {} & (env\ e; store\ s)store : ((bool\ b)store \rightarrow store : \\ & \quad (b \rightarrow \mathcal{C}'\{\!\!\{stmt_1\}\!\!\}(e))|(\mathcal{C}'\{\!\!\{stmt_2\}\!\!\}(e))) \\ & \qquad ((\mathcal{E}'\{\!\!\{exp\}\!\!\}(e)(s)) \mid bool)(s)\end{aligned}$$

$$\begin{aligned}\mathcal{C}'\{\!\!\{while\ exp\ do\ stmt\}\!\!\} = {} & rec(env\ e; store\ s)store : ((bool\ b)store \rightarrow store : \\ & \quad (b \rightarrow \mathcal{C}'\{\!\!\{stmt\}\!\!\}(e) \circ \mathcal{C}'\{\!\!\{while\ exp\ do\ stmt\}\!\!\}(e)) \\ & \qquad \mid ((store\ s')store : s'))((\mathcal{E}'\{\!\!\{exp\}\!\!\}(e)(s)) \mid bool)(s)\end{aligned}$$

4.4 Vienna Development Method

In this paper we have examined several models for verifying that a program meets its specifications. However, the real utility for any of these methods is their application to solving real-world software development problems. All of these methods are difficult to apply to realistically-sized programming tasks.

In order to address this, notations like Z (Spivey, 1988), Larch (Guttag, 1985), OBJ (Goguen, 1985), and VDM were developed as general purpose specification languages in addition to special purpose languages for specific applications such as: Anna for Ada programs (Luckham, 1985) or AS* for Pascal programs (Zelkowitz, 1991). In this paper we will discuss VDM developed by the IBM Vienna Laboratory (Jackson, 1985) (Jones, 1990). VDM, or the Vienna Development Method[3], adds very little to what has already been described, except, perhaps, for an arcane notation. What VDM does do, however, is to pick and choose from among the techniques we have studied, and to develop a formalism and presentation that lends itself to realistic problems. VDM has a much wider following in Europe today, and is being used on many difficult programming tasks.

VDM contains the following features:

- The basic concept is to define a specification of a program, define a solution to that specification and prove that the program meets the specification.
- It is based upon denotational semantics and Mills' functional correctness models in that it develops functions of the underlying program, and is concerned about transformations between program states.
- It permits formal proofs of program properties using a set of inference rules, similar to the Hoare rules we studied earlier.

In presenting a brief overview of VDM, let us revist our multiplication program $MULT$ that we developed earlier in Section 2.1.2. Using VDM we can give the specification to this program as follows:

$$\begin{array}{l} MULT : (A : \mathcal{N};\ B : \mathcal{N})\ y : \mathcal{N} \\ \quad \mathsf{ext}\ a, b : \mathsf{wr}\ \mathcal{N}; \\ \quad \mathsf{pre}\ B \geq 0 \\ \quad \mathsf{post}\ y = \overleftarrow{A}\overleftarrow{B} \end{array}$$

[3]VDM is not to be confused with the Vienna Definition language, or VDL, which was briefly described earlier.

This states that $MULT$ is a function with two arguments (A and B) of type $\mathcal{N}$(integer) that computes a result of type $\mathcal{N}$. It accesses state variables (i.e., global storage) a and b and writes to them. Access rd would mean read-only access to these variables.

This function has as a precondition $b \geq 0$ and as a postcondition $y = \overleftarrow{A}\overleftarrow{B}$, where $\overleftarrow{x}$ is the original value of x on entry to the function. Since we use pre- and postconditions very heavily, we can refer to them individually as pre$-MULT$ and post$-MULT$.

Any program that meets this specification must satisfy the following predicate:

$$\forall (i,j) \in \mathcal{N} \times \mathcal{N} \mid \mathsf{pre}{-}MULT(i,j) \Rightarrow MULT(i,j) \in \mathcal{N} \ \wedge \ \mathsf{post}{-}MULT(i,j,MULT(i,j)) \ \equiv$$
$$\forall (i,j) \in \mathcal{N} \times \mathcal{N} \mid j \geq 0 \Rightarrow MULT(i,j) \in \mathcal{N} \ \wedge \ MULT(i,j) = \overleftarrow{i}\,\overleftarrow{j}$$

First specify two functions $INIT$ and $LOOP$ whose composition gives us the above result. Their specification is as follows:

$$\begin{array}{l} INIT : (A : \mathcal{N};\ B : \mathcal{N})\ y : \mathcal{N} \\ \quad \mathsf{ext}\ a, b : \mathsf{wr}\ \mathcal{N}; \\ \quad \mathsf{pre}\ B \geq 0 \\ \quad \mathsf{post}\ a = \overleftarrow{A} \wedge b = \overleftarrow{B} \wedge y = 0 \wedge b \geq 0 \end{array}$$

$$\begin{array}{l} LOOP : y : \mathcal{N} \\ \quad \mathsf{ext}\ a, b : \mathsf{wr}\ \mathcal{N}; \\ \quad \mathsf{pre}\ b \geq 0 \wedge y = 0 \\ \quad \mathsf{post}\ y = \overleftarrow{a}\,\overleftarrow{b} \end{array}$$

We will use VDM to show that our previous solution meets this specification.

INIT function. We will define our candidate solution for $INIT$ using the same three initialization statements as before, using the symbol $\stackrel{\triangle}{=}$ as our function definition symbol:

$$INIT \ \stackrel{\triangle}{=}\ a \leftarrow A; b \leftarrow B; y \leftarrow 0$$

Our rules of inference are very similar to the Hoare axioms discussed previously; however, we want to make sure that each function terminates as part of the rule. For assignment, this is not a problem; potential problems only occur during loop statements.

While there are many such rules of inference, the two for assignment that we need are:

$$\begin{array}{ll} \textbf{asgn}_1 & \dfrac{}{\{true\}x \leftarrow e\{x = \overleftarrow{e}\}} \\ \textbf{asgn}_2 & \dfrac{x\ does\ not\ occur\ in\ E}{\{E\}x \leftarrow e\{E\}} \end{array}$$

We can prove pre$-INIT \Rightarrow$ post$-INIT$ via a proof similar to a Hoare-type axiomatic proof:

$\{B \geq 0\} a \leftarrow A \{B \geq 0\}$	*asgn*$_2$
$\{true\} a \leftarrow A \{a = \overleftarrow{A}\}$	*asgn*$_1$
$\{B \geq 0\} a \leftarrow A \{B \geq 0 \wedge a = \overleftarrow{A}\}$	*Consequence*
$\{B \geq 0 \wedge a = \overleftarrow{a}\} b \leftarrow B \{B \geq 0 \wedge a = \overleftarrow{a} \wedge b = \overleftarrow{B}\}$	*asgn*$_1$ & *asgn*$_2$
$\{B \geq 0\} a \leftarrow A; b \leftarrow B \{B \geq 0 \wedge a = \overleftarrow{a} \wedge b = \overleftarrow{B}\}$	*composition*
$\{B \geq 0\} a \leftarrow A; b \leftarrow B; y \leftarrow 0 \{B \geq 0 \wedge a = \overleftarrow{a} \wedge b = \overleftarrow{B} \wedge y = 0\}$	*asgn*$_1$ *asgn*$_2$ *composition*
$\{B \geq 0\} a \leftarrow A; b \leftarrow B; y \leftarrow 0 \{b \geq 0 \wedge a = \overleftarrow{A} \wedge b = \overleftarrow{B} \wedge y = 0\}$	*consequence*

LOOP function. We will define the same **while** statement to solve this *LOOP* specification:

$$LOOP \;\triangleq\; \textbf{while}\ b > 0\ \textbf{do begin}\ b \leftarrow b - 1; y \leftarrow y + a\ \textbf{end}$$

We need the following inference rule. If I is the invariant, T is the predicate on the **while** statement, and S is the loop body:

$$\frac{\{I \wedge T\}\ S\ \{I\}, \delta(T)}{\{I\}\textbf{while} \cdots \{I \wedge \neg T\}}$$

where $\delta(T)$ states that T is *wellfounded*, i.e. the predicate eventually becomes false and the loop terminates[4].

Using steps similar to the axiomatic proof in Section 2.1.2, we are able to complete the proof in a similar manner.

4.5 Use of VDM

This example, by no means, describes all of the features of VDM; however, the simple exercise should indicate the essential characteristics of a VDM development:

- It is better than most methods at separating the specification from implementation issues.
- It can be completely formal, but can also be used informally.
- It can be used at several levels – proving a design meets a specification, or proving that a source program meets the specs.
- While practical, the process is still not easy.

5 Multiattribute Specifications

It should now be clear that the specification of a software system depends more than simply verifying functionality. While we have focused on the verification attribute, others are often just as critical to the success of a project. While we cannot examine in depth all of these attributes in this paper, we survey some of them in order to provide the context for a complete software specification. We then complete this section with some recent research results which provide for an evaluation mechanism to choose among competing alternative solutions to a specification.

[4] Formally, we must show that there is no function f such that $f(i)$ *LOOP* $f(i+1)$ is a relation,, meaning that we eventually have to find some value of f outside of the domain of the test, and the test must then fail.

5.1 Resource usage

Of critical importance is the ability to judge how long and how expensive a given software system will take to build. Most techniques first divide a project into primitive components (the *work breakdown structure*) and then use a model to estimate each component of the system. This method works best when the individual pieces are well known and have been built previously.

Models to perform this analysis fall into two general categories: Those based upon environmental factors and those based upon source program complexity.

5.1.1 Environmental Factors

The concept of a *function point* (Albrecht, 1983) has been proposed as a mechanism to estimate development costs by counting the number of objects in a specification. Each file, module, data format, etc. forms a function point, and the number of function points can be related to the eventual cost of the system.

The general process to compute this measure is as follows:

1. *Basic Function Counts.* The number of external inputs, outputs and files are multiplied by a complexity factor (how complex program is estimated to be) in order to obtain a function count (FC) measure.
2. *Processing complexity.* Characteristics of the program (e.g., distributed functions, data communications, online access, reusability) are added together to get a degree of influence count.
3. *Function Counts.* Multiply together the above two factors using an appropriate scaling factor.

This gives an approximate size – in function points – which correlates fairly closely with lines of code measures and various cost and effort measures.

Boehm's Constructive Cost Model (COCOMO) (Boehm, 1981) is a more algorithmic approach to the function point model. Using the formula $Cost = a \times S^b$, where S is the size of the resulting source program, the model relates cost to the size and the calibration of the parameters a and b. Calibration is accomplished by tuning the model with fifteen cost drivers (e.g., product attributes such as reliability, computer attributes such as performance requirements, personnel attributes such as programmer capabilities and project attributes such as development methods used) which are assumed independent and multiplied together to obtain factor a.

COCOMO has been used successfully on many large projects. Its weakness, however, is that it depends upon program size, and these can only be estimated from the program's specifications.

5.1.2 Source Code Models

A second source of resource models depends more closely on the source program and the complexity of the program itself. These models generally follow a formula similar to the COCOMO model: $Cost = a \times S^b$ where the exponent b is slightly greater than 1. Environmental factors are not usually factored in; however, if multiple projects within a development environment of relatively homogeneous projects, most of the above environmental factors become part of the calibration process for a and b and this model becomes similar to COCOMO.

A final model to be presented is the Rayleigh curve, based upon hardware reliability theory. The assumptions for this model are that:

1. Software development consists of solving a fixed number of problems (i.e., source lines).
2. The next problem to solve is a random event based upon the number of available problems (e.g., size of the existing system).
3. This list of problems starts with 1 ("a spec"), grows to some maximum during development, and drops to 0 when the system is completed.

Using these assumptions, the solution becomes a Rayleigh curve giving the effort needed (unit cost) at time t:

$$Unit\ cost \ = \ 2Kate^{-at^2}$$

Developed initially by Putnam (Putnam, 1976), it has been shown to be an approximate measure on large systems (Basili, 1983).

5.2 Software Safety

With the increasing usage of computers to control real-time applications, the need to analyze risks in using these machine becomes vital. There is no such thing as absolute safety. Components break. And as much as we would not like to think about the dollar value of safety, the problem basically comes down to how much are will willing to pay for what level of safety?

High *reliability* usually means we produce a system that is failure-free. However, high *safety* means that the system is accident-free. This is related, but a different characteristic of a system.

With hardware, established mechanisms have long been developed for analyzing this problem. A complex system is broken down into smaller components, and the reliability for each is then measured. For example, a set of nails is "stressed" (i.e., bent) and after a point all will snap – each at a slightly different stress value. A statistical profile is then developed for that component. Based upon the level of performance needed, either that component can be used, or a more expensive component with higher stress values can be obtained.

Accidents rarely occur from single points of failures. These are usually handled by a specification. Some examples: Airplanes generally have three sets of cables connecting the pilot's controls to the flaps on the wings. In the U.S., 3 engine jets (e.g., Boeing 727, Lockheed L-1011) are all designed to fly on only one engine. Even if the faucet in your home sink fails, most building codes require a second valve in the wall. If that valve should also fail, then there is also the master valve controlling water into the house – even homes have triple redundancy.

As we move to fly-by-wire planes, it is necessary to now compute the reliability of this computer-software-hardware system. Specifications regularly require "seven 9s availability" (e.g., available 99.99999% of the time, or down 3 seconds per year). However, we have no real idea of how to compute these.

The analogy with hardware testing breaks down here. Hardware testing assumes components will break with some statistical accuracy. In the software case, if we test a program, then repeating the test will either *always* fail with that test data, or *never* fail. Each test will always be reapeatable.

However, the concept of *software fault tree analysis* by Leveson (Leveson, 1986) has been used to estimate software safety considerations. We briefly describe hardware fault free analysis and then give a brief summary of software fault tree analysis.

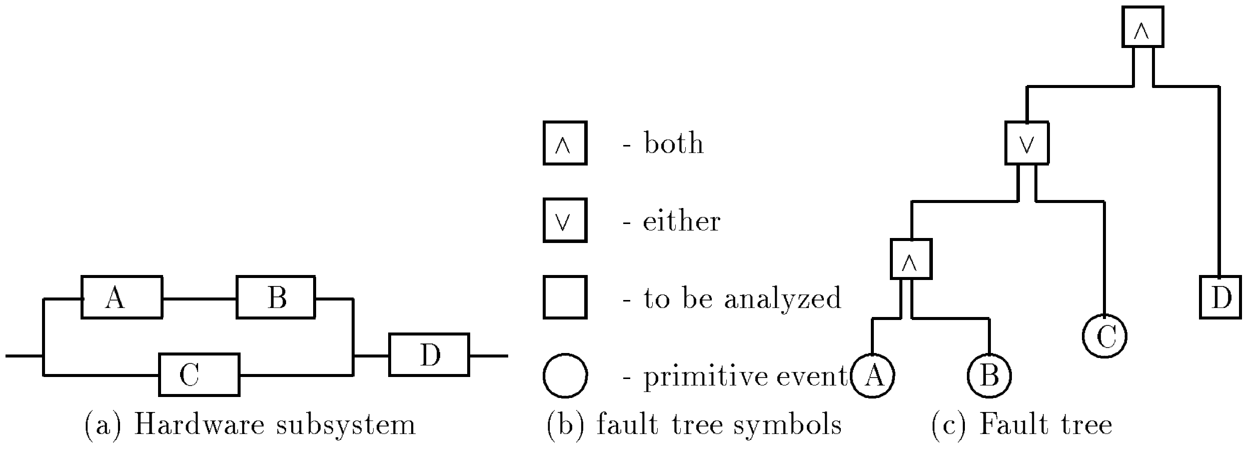

Figure 11: Fault tree analysis

5.2.1 Fault tree analysis

Consider the following simple problem. A system has two levels of protection – Subsystem A followed by B or subsystem C. If a new component D is added to the system, what is the probability of failure (Figure 11)?

Assume we know the probability of failure for components A, B and C. Since A and B are in series, failure of either of them results in system failure, and since C is in parallel to A and B, both paths have to fail for system failure.

from probability theory, we know that if $Pr(x)$ is the probability that component x fails, we have:

$$\begin{aligned} Pr(eitherA\ orB\ fails) = &\quad Pr(A) + Pr(B) - Pr(A) \times Pr(B) \\ Pr(A\ andB\ both\ fail) = &\quad Pr(A) \times Pr(B) \end{aligned}$$

Knowing these, we can compute the failure probability of the original subsystem using the *fault tree* in Figure 11(c). Call it $Pr(Subsystem)$. Since the new component D is in series with the original system, we compute the failure probability of the new system as $Pr(Subsystem) + Pr(D) - Pr(Subsystem) \times Pr(D)$. This then becomes a design constraint on how we need to design D to meet whatever reliability we require.

5.2.2 Software fault tree analysis

Assume Figure 11(a) represents a software design and there is a possible safety problem after component D executes. For a software system to cause an accident, the data will be in an incorrect state. The problem is to determine if the software can ever achieve that state from the initial state. Given the postcondition set of states at D, work backwards and determine which precondition set of states on D cause that problem. Continuing onward, determine the precondition set of states on the input to the program to determine if an accident state can ever occur.

What we have just described is the weakest precondition mechanism that Dijkstra developed for axiomatic verification. Thus we can view software fault tree analysis as just a special case of axiomatic verification – something we have dealt with extensively here.

5.3 Software design evaluation

While there are many other attributes which can affect a specification, their enumeration is beyond the scope of this paper. Suffice it to say that resource estimation and safety analysis are representative of the others.

Assume we have a set of such attributes which define a specification to a program. How does one choose an appropriate design that meets that specification? We know that a complete specification needs to address many attributes, including functionality, resource usage, security, safety and other concerns. Most designs will meet certain requirements well and probably not meet others.

How do we know what project to build? How do we assess the risk for each decision? Will prototyping reduce this risk? All of these are important considerations, and have only recently been applied to the specification problem (Charette, 1989).

We now describe one such evaluation strategy. It consists of two components: In one, the designer knows all relevant details (*decision under certainty*). In the second case, there is some variability (and risk) associated with all potential choices. We call this *decision under uncertainty*[5].

5.3.1 Decisions under certainty

We consider correct functionality to be just one of several attributes for a solution, with multiple designs implementing the same functionality. Lets first assume that our needed functionality is specified by a function (from state to state) and also the candidate programs are specified by functions from state to state. Let X be the functionality of program x. Program x is correct with respect to specification B if and only if $X \supseteq B$ (Mills, 1987a). We extend this model to include other attributes as well. Since these other attributes are often concerned with non-functional characteristics such as resource usage, schedules, and performance, we will use the term *viable* for any solution satisfying a specification rather that the more specific term *correctness*.

Now assume that our specifications (for both our needed software and the candidate programs) are vectors of attributes, including the functionality as one of the elements of the vectors. For example, X and Y are vectors of attributes that specify alternative solutions to a specification B. Let S be a vector of objective functions with domain being the set of specification attributes and range $[0..1]$. We call S_i a *scaling function* and it is the degree to which a given attribute meets its goal. We state that X $solves_S$ Y if $\forall i, S_i(X_i) \geq S_i(Y_i)$. We extend our previous definition of correctness to the following: design x is viable (i.e., is correct) with respect to specification B and scaling function vector S if and only if P $solves_S$ B. We can show that the previous definition of correctness is simply a one-dimensional example of this more general definition of viability (Cardenas, 1990).

Each attribute may not have the same importance. Assume a vector of weights W called *constraints* such that each $w_i \in [0..1]$ and $\sum w_i = 1$.

Our evaluation measure, the *performance level*, merges multiple scaled attributes and their constraints. Given specification vector X, scaling function S and constraints W, the performance level is given by: $PL(X, S, W) = \sum_i (w_i \times S_i(X_i))$.

We use the performance level as our objective function: Given a specification vector B, scaling vector S, constraints W and potential solutions x and y, X *improves* Y with respect to $\langle B, S, W\rangle$ if and only if:

[5]This development is based upon the recent research of Cárdenas and Zelkowitz (Cardenas, 1991)

1. X $solves_S$ B and Y $solves_S$ B
2. $PL(X,S,W) > PL(Y,S,W)$.

We use here a very simple weighted sum to compute the performance level. Our definition of *improves* depends only upon an appropriate definition of performance level for comparing two solutions, not on the details of how the two vectors are compared.

It should be noted that the model presented in this section depends upon the solution triple $\langle B,S,W\rangle$, which is a quantitative evaluation of how well each attribute of the proposed solution meets or exceeds the minimal specification B. We rarely know this in practice and this paper only assumes an ordinal ranking of the attributes—that is, one attribute value is better than another.

5.3.2 Decisions under uncertainty

We have so far assumed that the relative importance of each attribute is known a priori. However, we rarely know this with certainty. We therefore consider the following model, based upon aspects from economic decision theory (Boehm, 1981).

The performance level assumes that the relative importance of each attribute is a known constant, so the weight factors and scaling can be defined. However, this is not generally true. For example, in a program that includes a sort of a list of records, the importance of the sort algorithm itself depends upon how often it gets called and how long unsorted lists get. That is, if the list of items always remains short, then any sort algorithm will suffice since sorting will take a negligible part of the execution overhead. In this case, any attribute value (i.e., specification) describing the sort function will have minimal effect upon the resulting program and have a very low weight. Our problem is then to modify the previous model to account for unknowns in the importance for these attribute values.

Using terminology from decision theory, the potential solutions to a specification are called *alternatives*, and the various possibilities that will determine the importance for the attribute values are *states of nature*. Each state of nature is associated with a fixed set of weights giving the relative importance of each system attribute.

We can now represent the performance level as a matrix PL where $PL_{i,j}$ is the performance level for solution i under state of nature j. As before, the performance levels give a measure of how good a system is. We can approximate this by defining the entries $PL_{i,j}$ of performance level matrix PL as the payoff (e.g., monetary value) for solution i under state j. For example, assume we have two potential solutions X^1 and X^2, and assume we have 3 potential states of nature st_1, st_2 and st_3 which are represented as the 6 possible *payoffs* in the matrix:

$$PL = \begin{bmatrix} 100 & 500 & 0 \\ 300 & 200 & 200 \end{bmatrix} \tag{1}$$

In this example, if we knew for sure that st_2 would be the resulting state of nature, then we would implement alternative X^1 (with payoff 500), and if we knew that either states st_1 or st_3 were the resultant states, then alternative X^2 would be most desirable. However, we may not know this beforehand.

When the probability for each state of nature can be estimated, we can use expected values to achieve an estimated performance level. Given probability distribution vector P, where p_i is the probability that state of nature st_i is true, the expected payoff for alternative X^i is given by:

$$v_i = \sum_j pl_{i,j}\, p_j$$

Use the decision rule: Choose X^i which maximizes v_i or:

$$\max_i(\sum_j pl_{i,j}\, p_j)$$

For example, if we know that the probability distribution for each state of nature in our example is: P = (0.3, 0.5, 0.2), we can calculate the expected payoffs as follows:

$$\begin{aligned} v_1 &= 100 \times 0.3 + 500 \times 0.5 + 0 \times 0.2 \\ &= 280 \\ v_2 &= 300 \times 0.3 + 200 \times 0.5 + 200 \times 0.2 \\ &= 230 \end{aligned}$$

We would then choose X^1 over X^2 since $280 > 230$.

5.3.3 Risk aversion

Risk aversion plays an important role in decision making. This implies subjective behavior on the part of the software manager. We assume that the following *reasonable* behavior rule (i.e., equilibrium probability) is true:

- *Decomposition:* Given three payoffs $a \leq b \leq c$, there exists a probability ρ such that the decision maker is indifferent to the choice of a guarantee of b, and the choice of getting c with probability ρ and getting a with probability $1 - \rho$. We shall refer to this probability as $decomp(a, b, c)$.

For example, assume there are two techniques to solve a problem. One is fully tested giving a guaranteed payoff of \$5000 and a second new and more efficient technique promises a potentially larger payoff of \$10,000 (but not completely tested) with a chance to give a payoff of only \$2000. If a software manager considers using the new technique only if the chances of getting the payoff of \$10,000 are larger than 80%, the probability ρ is larger than 0.8. In this case the expected payoff will be $10{,}000 \times 0.8 + 2000 \times 0.2 = 8400$, so the given manager is somewhat risk-averse and conservative.

Let pl_0 be the minimal value in our payoff PL and let pl^* be the maximal value. In our example PL matrix (1), we would choose $pl^* = 500$ and $pl_0 = 0$. We decompose each $pl_{i,j}$ as $e_{i,j} = decomp(pl_0, pl_{i,j}, pl^*)$. This decomposition creates an equivalent pair of payoffs $\{pl_0, pl^*\}$, with probability $e_{i,j}$ of getting the more desirable pl^*. We call the matrix formed by these elements $e_{i,j}$'s the equilibrium matrix E.

Any element $e_{i,j}$ will satisfy the following inequality:

$$pl_0 \times (1 - e_{i,j}) + pl^* \times e_{i,j} \geq pl_{i,j}$$

The difference between the two sides of this equation reflects the manager's degree of risk averseness. If the two sides are equal, risk analysis reduces to the expected value.

5.3.4 Value of prototyping

Given the various unknowns in the states of nature, the software manager may choose to get more information with a prototype so that a better final decision can be made. However, before undertaking the procedure to extract more information, one should be sure that the gain due to the information will outweigh the cost of obtaining it. Here, we try to establish an absolute boundary: What is the value of perfect information?

The best we can expect is that the results of the experiment will indicate for sure which state of nature will hold. Under this case, we can choose the alternative that gives the highest performance level under the given state of nature:

$$\Phi = \sum_j p_j \times \max_i pl_{i,j}$$

In our example, we would choose X^1 under st_2 and choose X^2 otherwise, resulting in performance level Φ:

$$\begin{aligned}\Phi &= 0.3 \times 300 + 0.5 \times 500 + 0.2 \times 200 \\ &= 380\end{aligned}$$

What is the value of this perfect information? Since the expected value of our performance level was computed previously as 280, the value of this information is an improvement in performance level of $380 - 280 = 100$. This is the most that we can expect our prototype to achieve and still have it cost effective.

Assume we build a prototype to test which state of nature will be true. While we would like an exact answer, since a prototype is only an approximation to the real system, the results from prototyping are probabilistic. Let $result_1$, $result_2$, ... $result_k$ be the possible results of the prototype. This information will be presented in a conditional probability matrix C where $c_{i,j}$ represents the conditional probability of result $result_i$ given state of nature S_j.

Given the probabilities for each state (vector P) and the conditional probability matrix C, the marginal probability distribution (vector Q) for obtaining $result_i$ is given by:

$$q_i = \sum_j c_{i,j} \times p_j$$

We can compute the *a posteriori* distribution matrix P'. P' has as many rows as results from the prototype which are updated values of vector P. Row i gives the probabilities of the states of nature given that the result of the prototype is $result_i$:

$$p'_{i,j} = \frac{c_{i,j} \times p_j}{q_i}$$

Following our example, assume that a prototype of alternative X^2 is planned. The planned prototype can give the following results:

$result_1$. We are satisfied with the system as presented by prototype.

$result_2$. We are not satisfied.

Assume that the conditional probabilities are estimated beforehand. For example, we estimate that if the state of nature is st_2, we have probabilities 0.3 and 0.7 to obtain results $result_1$ and $result_2$ respectively from the prototype. The conditional probabilities appear in the matrix C having a column for each state and a row for each result of the prototype:

$$C = \begin{bmatrix} 0.9 & 0.3 & 0.4 \\ 0.1 & 0.7 & 0.6 \end{bmatrix}$$

From this we can calculate the probability q_i for each result i of the prototype, giving $Q = (0.5, 0.5)$, and the *a posteriori* distribution matrix:

$$P' = \begin{bmatrix} 0.54 & 0.30 & 0.16 \\ 0.06 & 0.70 & 0.24 \end{bmatrix}$$

If, for example, we get $result_1$ from the prototyping study, the new expected values for alternatives X^1 and X^2 are:

$$\begin{aligned} v_1 &= 100 \times 0.54 + 500 \times 0.3 + 0 \times 0.16 \\ &= 204 \\ v_2 &= 300 \times 0.54 + 200 \times 0.3 + 200 \times 0.16 \\ &= 254 \end{aligned}$$

In this case, alternative X^2 should be chosen, since it gives the higher performance level.

Similarly, if we should get $result_2$ from the prototyping study, the new performance levels for alternatives X^1 and X^2 are:

$$\begin{aligned} v_1 &= 100 \times 0.06 + 500 \times 0.7 + 0 \times 0.24 \\ &= 356 \\ v_2 &= 300 \times 0.06 + 200 \times 0.7 + 200 \times 0.24 \\ &= 206 \end{aligned}$$

In this case, alternative X^1 is the preferred choice.

Given that our expected performance level with no information was 280 (Section 5.3.2), we should only prototype if we gain from prototyping:

$$\begin{aligned} v_P &= 0.5 \times 356 + 0.5 \times 254 - 280 \\ &= 25 \end{aligned}$$

Since there is a positive gain $v_P = 25$, prototyping should be carried out as long as the cost to construct the prototyping study is less than this. Otherwise, an immediate decision should be made.

5.3.5 Implementation of model

A prototype implementation of this evaluation strategy has been built in C and runs on SUN 3 and DEC 3100 workstations. A manager enters a table of attributes and initial constraints and then executes the tool, called *Selector*. The manager is prompted for the various equilibrium probabilities which determine the risk averseness behavior of that particular individual, as well as objective characteristics of the particular solution being considered. The tool then computes the performance level for each potential solution, computes the potential gain from prototyping and offers advice on which attribute would provide the maximum gain if it were investigated.

6 Conclusion

The development of large software systems depends upon many factors. Correct functionality between a specification and a final product is one important criteria. In order to determine such correctness, three basic models of programming have been developed:

1. *A program is a logical object in mathematics.* With this view, we can view programming as the analog of a mathematical proof of a theorem (i.e., the specification). The axiomatic and predicate transformer techniques are based upon this model.
2. *A program is a mathematical function.* With this model, a program transforms some input domain into an output range of values. Function composition and various state-based transformations provide the mechanisms for proving correctness properties of programs. The functional and denotational semantics models are examples of this approach.
3. *A specification defines the relationship among the data objects in a program.* In this model, specifications determine algebraic properties among the components of the implementation. Algebraic data types is an example of this method of verification.

Much of this paper was devoted to giving details from three of these five models of program verification: Axiomatic, Functional and Denotational Semantics.

However, as stated in the introduction, a good specification consists of multiple attributes, with functionality being only one of them. In that light, we briefly surveyed some of the other models. We demonstrated:

1. Creation of a specification for a software system requires more than just determining the correct functionality for the system.
2. Showing that the functionality given by the specification; however, *is* an important component in system construction.
3. These techniques have been used to develop real systems. Tools and notations like Z and VDM have been developed to help build real systems.

Methods for choosing among competing designs have not been fully explored by the research community. Risk analysis and risk reduction is important in multi-million dollar efforts. We have described current research which addresses the needs for extended evaluation strategies that address the risks inherent in building complex software systems.

6.1 Future directions

The topics discussed in this paper are central to good system design and implementation. However, not all of the techniques have been commercialized and been adopted by industry. Current interest in the software engineering research community on these topics includes the following:

Verification. There are multiple formal models available for describing system functionality. Basic research in this topic probably will not progress much further in the near future. What is needed is the ability to transform these into practical algorithms. Work, like the development of VDM, needs to continue.

Specification attributes. Other than functionality, few algorithms exist for specifying systems. While there has been much work in the models and metrics area, measurement is still an imprecise art. Estimating resource needs and schedules is very ad-hoc today, even with formal models like COCOMO. Further work is needed here. The work on safety analysis is still in its infancy, and is tied into the verification work already described.

Risk analysis. While "risk" is a well established term, its application to the software problem is new. Further models are needed by software managers in order to determine development strategies before development begins.

Environments. None of the techniques can exist without machine support. As some of the examples in this paper have shown, proofs can be long and tedious. Tool support is crucial. Formal verifiers, implemented resource models, and risk analysis tools, like the prototype *Selector* all need to be developed in order to move these techniques from the state-of-the-art to the state-of-the-practice.

As given in this paper, there are many issues involved in specifying, building and verifying complex software systems. The technology has been developing over these past 25 years. While we have not fully automated the process, there are many techniques that can be applied to help address these important development issues.

7 Acknowledgements

Some of the research described in this paper has been partially funded by Air Force Office of Scientific Research grant 90-0031 to the University of Maryland. Much of the section on risk reduction was based upon work performed by the author and Dr. Sergio Cárdenas, now of AT&T Bell Laboratories.

8 References

(Albrecht, 1983) Albrecht A. J. and Gaffney J. E., Software function, source lines of code and development effort prediction: A software science validation, *IEEE Transactions on Software Engineering*, 9(6), 639–647.

(Basili, 1983) Basili V. R. and Zelkowitz M. V., Analyzing medium scale software development, In 3^{rd} *International Conference on Software Engineering*, pages 116–123, Atlanta, ACM and IEEE.

(Basu, 1975) Basu S. K. and Yeh R. T., Strong verification of programs, *IEEE Trans. on Software Engineering*, 1(3), 339–346.

(Boehm, 1981) Boehm B., *Software Engineering Economics*, Prentice Hall.

(Cardenas, 1990) Cárdenas S. and Zelkowitz M. V., Evaluation criteria for functional specifications, In 12^{th} *IEEE/ACM Int. Conf. on Software Engineering*, pages 26–33, Nice, FR.

(Cardenas, 1991) Cárdenas-Garcia S. and Zelkowitz M. V., A management tool for the evaluation of software designs, *IEEE Transactions on Software Engineering*, 17(9), 961–971.

(Charette, 1989) Charette R., *Software Engineering Risk Analysis and Management*, McGraw Hill.

(Craigen, 1990) Craigen D. (ed.), *Formal Methods for Trustworthy Computer Systems FM89*, Springer Verlag.

(Dijkstra, 1975) Dijkstra E. W., Guarded commands, nondeterminacy, and formal derivation of programs, *Communications of the ACM*, 18(8), 453–457.

(Floyd, 1967) Floyd R., Assigning meanings to programs, In *Proc.* 19^{th} *Symposia in Applied Mathematics*, pages 19–31. Amer. Mathematical Soc.

(Goguen, 1985) Goguen J. and Tardo J., *An introduction to OBJ: a language for writing and testing software specifications*, pages 391–420, Addison Wesley.

(Gries, 1981) Gries D., *The Science of Programming*, Springer Verlag, New York.

(Guttag, 1980) Guttag J., Notes on type abstraction (version 2), *IEEE Transactions on Software Engineering*, 6(1), 13–23.

(Guttag, 1978) Guttag J. and Horning J. J., The algebraic specification of abstract data types, *Acta Informatica*, 10, 27–52.

(Guttag, 1985) Guttag J., Horning J. J. and Wing J. M., The Larch family of specification languages, *IEEE Software*, 2(5), 24–36.

(Hall, 1990) Hall A., Seven myths of formal methods, *IEEE Software*, 7(5), 11–19.

(Hoare, 1969) Hoare C. A. R., An axiomatic basis for computer programming, *Communications of the ACM*, 12(10), 576–580, 583.

(Jackson, 1985) Jackson M. I., Developing Ada programs using the Vienna Development Method (VDM), *Software Practice and Experience*, 15(3), 305–318.

(Jones, 1990) Jones C. B., *Systematic Software Development using VDM*, Prentice Hall.

(Knuth, 1968) Knuth D. E., Semantics of context-free languages, *Mathematical Systems Theory*, 2, 127–145.

(Leveson, 1986) Leveson N., Software safety: Why, what and how, *ACM Computing Surveys*, 18(2), 125–163.

(Lucas, 1969) Lucas P. and Walk K., On the formal description of PL/I, *Annual Review in Automatic Programming*, 6, 105–182.

(Luckham, 1985) Luckham D. C. and von Henke F. W., Overview of Anna, a specification language for Ada, *IEEE Software*, 2(2), 9–23.

(Marcotty, 1976) Marcotty M., Ledgard H. F., and Bochman G. V., A sampler of formal definitions, *ACM Computing Surveys*, 8(2), 192–198, 232–250.

(Mills, 1987a) Mills H., Basili V., Gannon J., and Hamlet R., *Principles of computer programming: A mathematical approach*, Allyn Bacon.

(Mills, 1987b) Mills H. D., Dyer M., and Linger R. C., Cleanroom software engineering, *IEEE Software*, 4(5), 19–25.

(Musser, 1979) Musser D. R., Abstract data type specifications in the affirm system, In *IEEE Specifications of Reliable Software Conference*, pages 47–57, Cambridge MA.

(Musser, 1980) Musser D. R., On proving inductive properties of abstract data types, In *Proc. 7th ACM Symposium on Principles of Programming Languages*, pages 154–162, Las Vegas.

(Putnam, 1976) Putnam L., A macro-estimating methodology for software development, In *Compcon*, pages 138–143. IEEE Computer Society.

(Scott, 1971) Scott D. and Strachey C., Towards a mathematical semantics for computer languages, In *Symp. Computers and Automata*, pages 19–46. Polytechnic Inst. of Brooklyn.

(Spivey, 1988) Spivey J. M., *Introducing Z: A specification language and its formal semantics*, Cambridge University Press.

(Wing, 1990) Wing J. M., A specifier's introduction to formal methods, *IEEE Computer*, 23(9), 8–24.

(Zelkowitz, 1991) Zelkowitz M. V., and Càrdenas S., The role for executable specifications in system maintenance, *Information Sciences Journal*, 57, 347–359.