SURVEY & TUTORIAL SERIES

A Functional Correctness Model of Program Verification

Marvin V. Zelkowitz
University of Maryland

Translating a problem description into a computer solution is central to programming, but the process is generally ill-defined, error-prone, and expensive. Some researchers are using formal mechanisms to describe a problem and algorithmic processes to convert the problem statement into a program. The general approach is to describe a problem in a specification language with well-defined syntax and semantics. This reduces the problem to developing a source program that meets the specification. Among the specification languages investigated, axioms[1] and algebraic models[2] predominate.

Verification is just one of many approaches to producing quality software. (The sidebar on the following page shows where it fits in the overall scheme.) The functional approach described in this article is one alternative, but it is not meant as a panacea for poor requirements and specifications analysis. Regardless of the model employed, verification is a precise, formal, and difficult undertaking. Some applications do, however, lend themselves to a functional approach that has not been adequately described in the literature.

The University of Maryland uses a functional correctness model as part of its introductory computer science course. The model was originally developed by Mills,[3] who, with others,[4-6] has since refined it. The idea is to express a specification as a mathematical function, develop a program, and prove that the function implemented by that program is the same as the specification function. The system used at the university meets several of Dijkstra's criteria[7] for the teaching of formalism to express programs.

This model's verification conditions depend only on elementary symbolic execution of a trace table. An easy-to-learn technique, it's used in a freshman computer science course.

In this article, the method is applied to rather simple programs. However, even in large complex implementations, the techniques can be applied informally to determine the functionality of complex interactions.

Functional model of a program

Specifications. A specification is a mathematical description of a problem to be solved. Let α be a string representing a source program. For example, a Pascal program is just the linear string

program main(input, output); ... end.

We express the mathematical function denoted by program α by a *box notation.** $[\alpha]$ represents the function that com-

*[p] is often written as $\boxed{p}$ in other papers on the subject.

0018-9162/90/1100-0030$01.00 © 1990 IEEE

Approaches to producing quality programs

Testing. In the oldest technique, testing, programs are executed using sample data that is representative of the data processed under actual use. If the data is chosen appropriately, most errors can be found. But, as Dijkstra has observed, testing can only show the presence of bugs, not their absence. In most large implementations, testing is the most feasible and generally the only usable technique.[1]

Design methodology. Good techniques produce well-structured programs, which minimize faulty logic and hence errors. Techniques like structured programming, data abstractions, top-down design, and object-oriented programming help the programmer think more clearly about the programming process. While these methods are great aids in producing quality programs, the programs must still be checked using other techniques for the eventual programming glitch.[2]

Verification. Under this technique, programs are viewed as formal objects developed from a set of precise specifications. Once developed, they are guaranteed to produce the output given in the specifications. This is the model developed in this article. Although it guarantees the stated output, the technique has problems: there is no guarantee that the specifications are correct, and the development of proofs is extremely difficult. The major techniques are *axiomatic*, where programs are considered extensions to the predicate calculus[3]; *algebraic*, which views programs as equations[4]; and *functional*, the approach in this tutorial.

Reliability. Reliability is the probability that software will not cause a system failure for a specified period of time, whether or not the data presented to the software meet the program's specifications. This contrasts to the above definition of verification. Correct programs may be very unreliable, and reliable programs may not be correct. For example, consider two watches — one stopped and the other two hours late. The stopped watch is correct twice a day; the late watch is never correct. However, the stopped watch is highly unreliable, while the late one is quite reliable.

Testing is often the best method to show good reliability.[5] *Software safety* is a related topic that addresses reliability and the probability of embedded systems causing physical harm to individuals.[6]

As can be seen by the above list, all of the techniques are useful, but they are difficult to use effectively. Improving these methods is a major focus of software engineering research.

References

1. R. DeMillo et al., *Software Testing and Evaluation*, Benjamin Cummings, Menlo Park, Calif., 1987.
2. E. Yourdon, *Writings of the Revolution: Selected Readings on Software Engineering*, Yourdon Press, New York, 1982.
3. D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.
4. J. Guttag, "Notes on Type Abstraction (Version 2)," *IEEE Trans. Software Eng.*, Vol. 6, No. 1, Jan. 1980, pp. 13-23.
5. D.L. Parnas, J. van Schowen, and S.P. Kwan, "Evaluation of Safety-Critical Software," *Comm. ACM*, Vol. 33, No. 6, June 1990, pp. 636-648.
6. N. Leveson, "Software Safety: Why, What and How," *Computing Surveys*, Vol. 18, No. 2, 1986, pp. 125-163.

putes 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. In choosing one optimal strategy from several equivalent ones (for example, 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*[5]:

> Program p is correct with respect to specification relation r if and only if domain($r \cap$ [p]) = domain(r).

In other words, if we take the subset of r from those pairs in r that are also in function [p] (that is, $r \cap$ [p]), we have a function. If this function has the same domain as r, then [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 use the simpler case. We have chosen the more restricted specification function f, instead of the more general relation r, with the corresponding correctness theorem of $f \subseteq$ [p].

Programs. 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 denotational semantics, we can define the meaning of such a program as follows:

> If *var* is a set of variable names and *val* is a set of values, a *state* is a function with the signature *state* : *var* $\rightarrow$ *val*. A state represents the formal model for program storage (for example, activation records).
>
> If *expr* is an expression, [expr] is a function that maps a state into values, or [expr] : *state* $\rightarrow$ *val*. For example, if (x,a) and (y,b) represent entries in the state function S representing variables x and y, then [x+y] (S) is defined to be the function with [x] (S) + [y] (S) as a value. If we define [x] (S) to be $S(x)$, then $S(x) = a$, which agrees with our intuitive definition that [x+y] (S) = a+b.
>
> If s is a Pascal statement, then [s] is a function that maps a state into a state — that is, 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 (var,val) pair for the newly declared variable. For example, if s is the state $\{(x,1),(y,2)\}$, then the function [y:=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 a sequence $s_1, s_2, \ldots, s_n$ of statements, then [s] = [s_1, s_2, ... , s_n] = [s_1] $\circ$ [s_2] $\circ$ ... $\circ$ [s_n] = [s_n] (... ([s_2] ([s_1])) ...).

The function [p] for "program main (input, output); begin s1; s2; ... end." is given by [program main (input, output)] $\circ$ [s_1] $\circ$ [s_2] $\circ$ ... $\circ$ [.] where the signature for [program...] is *val* $\rightarrow$ *state*, for [.] is *state* $\rightarrow$ *val*, 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.)

Developing a program requires several separate activities:

(1) designing a specification that expresses the task to be performed,
(2) refining that specification into a formal explicit statement that captures the specification's intended functionality, and
(3) developing a program that correctly implements that functionality.

Most of this article concerns the transition between the last two steps. Techniques will be given that aid in this transition and help show that both formalisms have equivalent functionality.

Applications. With this notation, three separate activities — verification, program design, and reverse engineering — can be investigated:

(1) If f is a function and if p is a program, show $[p] = f$ — that is, verification.

(2) If f is a function, develop program p such that $[p] = f$ — that is, program design. As a practical matter, we only care that $f \subseteq [p]$, since any value in $[p]$ and not in f represents a value computed by the program that is outside its specifications and not of interest to us.

(3) If p is a program, then find a function f such that $[p] = f$ — that is, reverse engineering. Given a program, determine its specifications. Some heuristics are given, but the basic method is to "guess" a solution and show by methods 1 and 2 above that it is the correct solution.

Symbolic execution

Symbolic execution is an aid in showing functional composition. To show that $[p] = f$, we symbolically execute program p and show that the resulting function is the same as f.

For example, consider the Pascal sequence

```
x:= x+1;
y:= x+y;
x:= y+1
```

Since we know that

$$[x:=x+1;\ y:=x+y;\ x:=y+1] = [x:=x+1] \circ [y:=x+y] \circ [x:=y+1]$$

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. In the statement function, we substitute the value of each variable at that point in the computation. This results in a new function that can transform each variable into its new value.

For the above Pascal sequence, we get the following trace table:

Part	x	y
x:= x+1	x+1	
y:= x+y		(x+1)+y
x:= y+1	(x+1+y)+1 = x+y+2	

This states that simultaneously x is transformed by the function [x:=x+y+2] and y is transformed by [y:=x+y+1].

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

For example, the program sequence

```
x:= x+y;
if x>y then
    x:= x−1
```

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

Part	Condition	x	y
x:= x+y		x+y	
if x>y	(x+y)>y		
x:=x−1		(x+y)−1	

and

Part	Cond.	x	y
x:= x+y		x+y	
if x>y	(x+y)≤y		

These two tables represent the following: if $x+y>y$, the function is [x:= x+y−1]; if $x+y \leq y$, the function is [x:= x+y]. The next section shows how to write this as a conditional assignment function.

Design rules

As stated earlier, software development consists of (1) designing the specification, (2) formalizing the specification, and (3) developing the source program. We use a functional notation for step 2 that is closely tied to the eventual Pascal source program. This notation includes (1) concurrent assignment, (2) conditional assignment, and (3) loop verification. This notation was strongly influenced by McCarthy's work on Lisp.

Designing assignment statements. Concurrent assignment is defined as simultaneous assignment. The function

```
(x,y,z := 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, the state function that results will have the same values for all state variables other than x, y, and z, and those three will have new values.

Given statement p, showing that $[p]$ 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 (that is, all values needed by a left-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 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 (that is, when finished).

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

Remember, however, that the three design rules are heuristics, not an algorithm. They indicate how to search for a solution and how to check if the solution is correct, but they do not give the solution. We have not replaced the art of programming by an implementable methodology that automatically builds correct programs from specifications.

Designing 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

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

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 that 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 (that is, the expression *true*), then it can be omitted, with the last term becoming (c_n). The *identity* function is written as ().

We'll use several theorems involving conditional statements in this article. They can be verified by simple trace tables:

(1) Conditional $(a \rightarrow b) \mid (\text{not}(a) \rightarrow c)$ has the same meaning as $(a \rightarrow b) \mid (c)$.
(2) Conditional $(a \rightarrow (b \rightarrow c))$ has the same meaning as (a and $b \rightarrow c$).
(3) Conditional $(a \rightarrow c) \mid (b \rightarrow c)$ has the same meaning as (a or $b \rightarrow c$).
(4) Conditional $(a \rightarrow (b \text{ or } c))$ has the same meaning as $(a \rightarrow b) \mid (a \rightarrow c)$.

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

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

the Pascal program can be written directly as

```
if b1 then c1
else if b2 then c2
else if b3 then c3
...
```

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

Verifying 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, ... are the conjunctions of all conditions on each execution path, and a_1, a_2, ... are the corresponding concurrent assignments. The function f that this implements is

$$(p_1 \rightarrow a_1) \mid (p_2 \rightarrow a_2) \mid \ldots \mid (p_n \rightarrow a_n)$$

Consider the following example:

```
x:=x+y;                 (* 1 *)
y:=y-x;                 (* 2 *)
if x+y>0 then
        y:=x+y   (* 3 *)
    else
        y:=-x-y  (* 4 *)
```

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

(1) If is true:

Part	Cond.	x	y
x:=x+y		x+y	y
y:=y-x		x+y	-x
if x+y>0	(x+y)-x>0		
y:=x+y			x+y-x=y

(2) If is false, so not(if) is true:

Part	Cond.	x	y
x:=x+y		x+y	y
y:=y-x		x+y	-x
if x+y>0	(x+y)-x≤0		
y:=-x-y			-(-x) -(x+y) =-y

This gives the function:

$$(y>0 \rightarrow x,y := x+y,y) \mid (y \leq 0 \rightarrow x,y := x+y,-y)$$

Or, since the assignment to y (that is, $(y>0 \rightarrow y := y)$ and $(y \leq 0 \rightarrow y := -y)$) is just function abs($y$), the function reduces to

$$(x,y := x+y, \text{abs}(y))$$

We could have left our answer as a conditional assignment, but 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 any encountered in axiomatic verification.

Verifying while loops. 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 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 V.I-V.

The while statement [while b do d] is defined recursively via the if statement to mean[5]

[while b do d] = [if b then begin d; while b do d end]

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

(**) [while b do d] = [if b then d; while b do d] = [if b then d] ∘ [while b do d]

Let f be the meaning of the while statement, that is, f = [while b do d]. By substituting back into (**) above, we get the first condition that

(V.I) f = [if b then d] ∘ f

What other conditions on f ensure 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. To ensure that this cannot happen, we require that f be defined whenever [while] is defined, or that domain([while]) ⊂ domain(f). (Note: For ease in reading, we will use [while] to stand for [while b do d]).

Similarly, if [while] is everywhere the identity function, then any f will fulfill the equation since the recursive equation reduces to f = () ∘ f = f. Thus, we must also have domain(f) ⊂ domain([while]). This yields

(V.II) domain(f) = domain([while b do d])

Consider any state $s \in$ domain([while]). If [b] (s) is true, that is, expression b in state s is true, then from (**), s_1 = [d] (s) and $s_1 \in$ domain([while]). This will be true, for s_2, s_3, and so on, until at some point [b] (s_n) is false and both [if b then d] (s_n) and [while b do d] (s_n) equal s_n.

This s_n is a member of domain([while]) and of range([while]). More importantly, if [b] (s) evaluates to false, then [while] (s) = s. Or, stated another way, [while] (s) = s for all states s where [b] (s) is false. This is just a restriction on the [while] function to those states where b is false, which is the function (not(b) → [while]). This must be

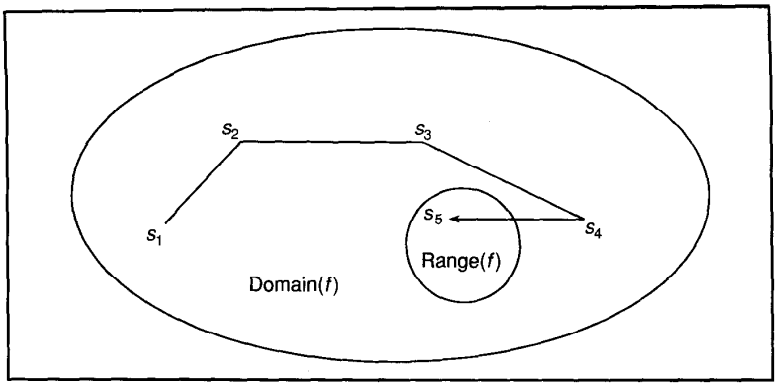

Figure 1. Domain and range of the while function.

equal to the identity function (), also restricted to the same domain, or just (not(b) $\rightarrow$ ()). Any candidate function f must also have this property, yielding the third constraint

(V.III) (not(b) $\rightarrow$ f) = (not(b) $\rightarrow$ ())

Designing 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 [b] evaluates to false, and range([while]) is just the set of states where [b] is false. But since we can apply [while] to this state initially, it is also part of domain([while]). Therefore, as shown in Figure 1, range([while]) $\subset$ domain([while]). Since f must also have this property, we get the first design constraint:

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

Similarly, we have shown that for an s where [b] (s) is false, [while] (s) = s, we must also have $f(s) = s$, because if [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) [b] evaluates to true in domain(f)-range(f) and false in range(f).

Verification example using the functional correctness model

This example shows that the function $f = (A \leq B \rightarrow A, B := (B-(B-A)/2), (B-(B-A)/2) \mid ())$ is implemented by the source program

```
1   while A<B do
2     begin
3     A := A+1;
4     if A<B then
5       B := B-1
6     end
```

where A and B are integers and division means integer truncated division (for example, 1/2 = 0).

The approach we follow is to first determine the functionality of the assignment statement (line 3), then the if statement (lines 4-5), then the entire begin block (lines 2-6), and finally the functionality of the entire segment (lines 1-6).

Line 3. A := A+1 is just the concurrent assignment ($A := A+1$).

Lines 4-5. d_1 = [if A<B then B := B–1]

If $A<B$ is true, evaluate the function $B := B-1$, and if it is false, skip the then statement and do nothing, for example, the identity function. The conditional assignment can be written as

$d_1 = (A<B \rightarrow B := B-1) \mid ()$

Lines 2-6. d_2 = [begin A := A+1; if A<B then B := B–1 end]

d_2 = [A:=A+1] $\circ$ [if A<B then B:=B–1] = ($A:=A+1$) $\circ$ d_1
= ($A:=A+1$) $\circ$ (($A<B \rightarrow B:=B-1$) | ())

Develop a trace table for the begin block. There will be two paths through this block (for example, first and second alternatives for d_1). Hence, there will be two trace tables:

Part	Cond.	A	B
3: A:=A+1		A+1	
4: if A<B	(A+1)<B		
5: B:=B–1			B–1

and

Part	Cond.	A	B
3: A:=A+1		A+1	
4: if A<B	(A+1)≥B		

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, that is, the loop must terminate for all $x \in$ domain(f).

Given function f, we develop a while statement such that [while] = f as follows:

(1) *Existence.* Verify conditions D.I and D.II. If these cannot be satisfied, then no such while statement can be written.

(2) *Range determination.* Use D.III to develop some predicate b such that [b] is false on range(f) and true on domain(f)-range(f). Since f and [while] are to be the same function, b becomes the predicate for the loop.

(3) *Loop body.* Use D.IV to develop an appropriate d. These guidelines do not give absolute solutions to this problem, but they do indicate how to verify whether d, once found, is a solution. It is comparable to finding the loop invariant in an axiomatic proof.

(4) *Termination.* Prove that the selected b and d cause the loop to terminate (condition D.V). If proven, since step 2 shows that [b](x) is false for $x \in$ range(f), this shows that the loop will terminate with some x in this range.

Examples. For two simple examples of this method, see the sidebars below and on page 38. The first example verifies a program with its specifications; the second example shows the design of a program from its functional specification. For a more complex example, see Gannon, Hamlet, and Mills.[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.

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.

Isolation of the type definition and invocation of the procedures that operate on such objects are crucial to the data abstraction model. 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 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.

We then get

$$d_2 = (A+1<B \rightarrow A,B := A+1,B-1) \mid (A := A+1)$$

Lines 1-6. Show f = [while A<B do begin A:=A+1; if A<B then B:=B−1 end]

We must show that function f meets the three verification rules. We will do this in the order V.II, V.III, and V.I.

(1) *Show V.III.* (not($A<B$) $\rightarrow f$) = (not($A<B$) $\rightarrow$ ( ))

(not($A<B$) $\rightarrow f$) =

($A \geq B \rightarrow$ ($A \leq B \rightarrow A,B := B-(B-A)/2, B-(B-A)/2$) | ( )) =

($A \geq B$ and $A \leq B \rightarrow (A,B := B-(B-A)/2, B-(B-A)/2)$) | ($A \geq B \rightarrow$ ( )) =

($A=B \rightarrow A,B := B-(B-A)/2, B-(B-A)/2$) | ($A \geq B \rightarrow$ ( )) =

($A=B \rightarrow A,B := A,B$) | ($A \geq B \rightarrow$ ( )) =

($A \geq B \rightarrow$ ( ))

(2) *Show V.II.* domain(f) = domain([while])

f is defined for all A and B. For $A \leq B$, an explicit assignment is given, and for all other A and B, f is the identity function.

The [while] function is also defined for all A and B. If $A \geq B$, the body of the while does not execute giving the identity function for such A and B. If $A<B$, then for each pass through the loop, A is increased by 1 and B may be decremented by 1. At some point, $B-A$ must reach 0 or become negative. If $B-A \leq 0$, then $B \leq A$ and the while loop terminates. So for all A and B, the while statement must terminate and will generate some value for A and B.

(3) *Show V.I.* f = [if b then d] $\circ f$

The meaning of the body of the if statement (d) is the previously defined function:

$$d_2 = (A+1<B \rightarrow A,B := A+1,B-1) \mid (A := A+1)$$

The problem then reduces to showing that

$$f = [\text{if A<B then } (A+1<B \rightarrow A,B:=A+1,B-1) \mid (A:=A+1)] \circ ((A \leq B \rightarrow A,B := B-(B-A)/2, B-(B-A)/2) \mid ( ))$$

We will generate the set of functions that represent each separate path through each possible trace table. If we let c_1 be the if expression $A<B$, c_2 be $A+1<B$ in d_2, and c_3 be $A \leq B$ in f, then there are six possible paths through this function yielding six different trace tables, each deriving a different function g_i:

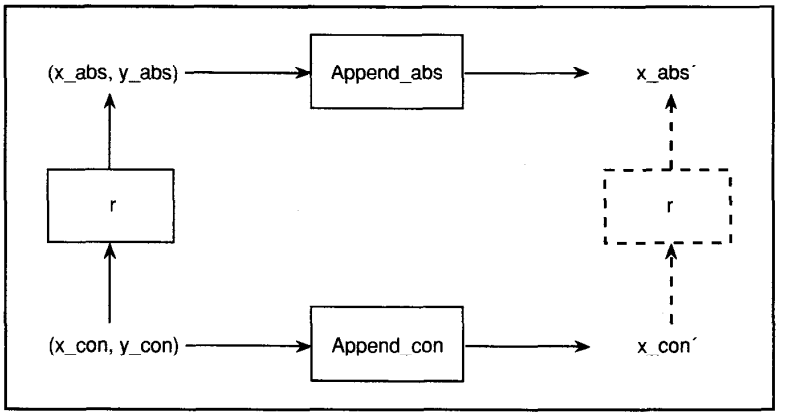

Figure 2. Commuting representation diagram.

Languages such as Ada (or C++) allow 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 procedure names contained in the module.

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

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 only defines fixed-length strings; therefore, we must implement this as objects using primitive Pascal data types.

In procedures outside the defining module, we would like to refer to these objects (for example, 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 *concrete functions* that give the implementation details.

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

---

c_1	c_2	c_3	Function
true	true	true	g_1
true	false	true	g_2
true	true	false	g_3
true	false	false	g_4
false	—	true	g_5
false	—	false	g_6

(Note: If c_1 is false, then d_2 is not evaluated, giving only six possibilities rather than the full complement of eight that normally occurs with three predicates.)

We need to show that

$f = g_1 \mid g_2 \mid g_3 \mid g_4 \mid g_5 \mid g_6$

For g_1:

Part	Cond.	A	B
c_1 is true	A<B		
c_2 is true	A+1<B	A+1	B–1
c_3 is true	A+1≤B-1	B–1–(B–1–(A+1))/2= B–(B–A)/2	B–1–(B–1–(A+1))/2= B–(B–A)/2

The resulting predicate, $(A<B)$ and $(A+1<B)$ and $(A+1\leq B-1)$, reduces to $A<B-1$; $g_1 = (A<B-1 \rightarrow A,B := B-(B-A)/2, B-(B-A)/2)$

For g_2:

Part	Cond.	A	B
c_1 is true	A<B		
c_2 is false	A+1≥B	A+1	
c_3 is true	A+1≤B	B–(B–(A+1))/2	B–(B–(A+1))/2

The resulting predicate is $(A<B)$ and $(A+1\geq B)$ and $(A+1\leq B)$, which reduces to $A=B-1$.

By substituting $B-1$ for A, we get

$$g_2 = (A=B-1 \rightarrow A,B := B-(B-(A+1))/2, B-(B-(A+1))/2)$$

$$= (A=B-1 \rightarrow A,B := B-(B-(B-1+1))/2, B-(B-(B-1+1))/2)$$

$$= (A=B-1 \rightarrow A,B := B,B)$$

However, if $A=B-1$, then $(B-A)/2=0$. Thus we can write g_2 as

$$g_2 = (A=B-1 \rightarrow A,B := B-(B-A)/2, B-(B-A)/2)$$

For g_3:

Part	Cond.	A	B
c_1 is true	A<B		
c_2 is true	A+1<B	A+1	B–1
c_3 is false	A+1>B–1		

{abs: $x\_abs = <x_1, x_2, ..., x_n>$}

The concrete representation of a Vstring could be

```
{con: x_con: record
   chars: array [ 1...maxval] of char;
   size: 0..maxval
   end:}
```

To show that both representations are the same, we define a *representation function* that maps concrete objects into abstract objects. It does this 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 (that is, the set of variables defined by the above record description) and Vstrings is the set of abstract strings, then we define a representation function r with the signature

r: state → state

such that $r = \{(u,v) \mid u=v$, except that if $u(x) \in$ Cstrings, then $v(x) \in$ Vstrings$\}$. This simply means that u and v represent the same set of variables in the program store, except that each occurrence 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:

$x\_abs:=<x\_con.chars[i] \mid 1 \le i \le x\_con.size>$

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:=Append (x,y) as

{abs: $x_1,..., x_n,..., x_{n+m} := x_1,..., x_n, y_1,..., y_m$}

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

```
{con: x.chars[n+1],...,
   x.chars[n+y.size],
   x.size := y.chars[1],..., y.chars[y.size],
   x.size+y.size}
```

If x_con and y_con represent the concrete implementations of Vstrings x and y, and if x_abs and y_abs represent their abstract representation, and if Append_con and Append_abs represent the concrete and abstract functions, we have

x_con' := Append_con (x_con, y_con)
x_abs' := Append_abs (x_abs, y_abs)

We want to know if both the concrete and abstract functions achieve the same result, or if the abstract representation of what we get by implementing Append_con is 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 2 *commutes* (that is, 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 (for example, $x_1, x_2,..., x_n, y_1,..., y_m$).

As given by our earlier correctness the-

This leads to the condition $(A<B)$ and $(A+1<B)$ and $A+1)>(B-1)$. We get $(A<B-1)$ and $(A>B-2)$, which is the null function.

For g_4:

Part	Cond.	A	B
c_1 is true	A<B		
c_2 is false	A+1≥B	A+1	
c_3 is false	A+1>B		

The resulting condition is $(A<B)$ and $(A+1\ge B)$ and $(A+1>B)$. But $(A<B)$ and $(A+1>B)$ are mutually disjoint, making g_4 null.

For g_5:

Part	Cond.	A	B
c_1 is false	A≥B		
c_3 is true	A≤B	B–(B–A)/2	B–(B–A)/2

The resulting condition is $(A\ge B)$ and $(A\le B)$ or $A=B$. For $A=B$, $B-(B-A)/2 = B = A$; $g_5 = (A=B \rightarrow A, B := A,B)$.

For g_6:

Part	Cond.	A	B
c_1 is false	A≥B		
c_3 is false	A>B	A	B

The resulting condition is $(A\ge B)$ and $(A>B)$ or just $(A>B)$; $g_6 = (A>B \rightarrow A,B := A,B)$.

Next, show $f = g_1 \mid g_2 \mid g_3 \mid g_4 \mid g_5 \mid g_6$. In this example, since g_3 and g_4 are null, we have to show that $f = g_1 \mid g_2 \mid g_5 \mid g_6$.

$g1 \mid g2 \mid g5 \mid g6 =$
$(A<B-1 \rightarrow A,B := B-(B-A)/2, B-(B-A)/2) \mid$
$(A=B-1 \rightarrow A,B := B-(B-A)/2, B-(B-A)/2) \mid$
$(A=B \rightarrow A,B := A,B) \mid$
$(A>B \rightarrow A,B := A,B)$

The first two terms reduce to

$(A<B \rightarrow A,B := B-(B-A)/2, B-(B-A)/2)$

For $A=B$, the third term becomes

$(A=B \rightarrow A,B := A,B) =$
$(A=B \rightarrow A,B := B-(B-A)/2, B-(B-A)/2)$

And the last term is

$(A>B \rightarrow A,B := A,B) = (A>B \rightarrow ())$

We have therefore shown that

$g1 \mid g2 \mid g5 \mid g6 =$
$(A\le B \rightarrow A,B := B-(B-A)/2, B-(B-A)/2) \mid (A>B \rightarrow ()) =$
$(A\le B \rightarrow A,B := B-(B-A)/2, B-(B-A)/2) \mid () =$
f

orem, a program (for example, Append_con) will often compute a value in a domain larger than necessary (for example, domain(Append_abs)). Thus, we actually want to show

$$r \circ \text{Append\_abs} \subseteq \text{Append\_con} \circ r$$

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 translatable into a Pascal source program via the techniques described.

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.

Note the order of steps 2 and 3. It is important to understand the abstract functions before designing the concrete representation, since the appropriate representation will depend greatly on the application. Consider the implementation of a *date* data object. Depending on the abstract functions required, the following are all feasible concrete representations:

(1) *Store as character string MM/DD/YY.* This is appropriate if the date is simply a unique tag associated with some data and has no other semantic meaning.

(2) *Store as <YY,DDD> where integer YY is the year and integer DDD is the day of year.* This is quite efficient if sequential dates are needed.

(3) *Store as number of days since some initial date.* This is most efficient to compute distances between two days, avoids certain problems such as accounting for leap years in all functions, but is cumbersome to print out in its usual format.

(4) *Store as <MM,DD,YY> for integers MM, DD, and YY.* Computation on dates is a bit more cumbersome, but conversion to its usual printed form is quite easy.

The importance of this technique is that it can be applied at any level of detail. This article obviously considered only short program segments. For larger programs, only concepts critical to the success of a program need to be formalized, although a long-range goal would be to develop this or other techniques that could be applied to very large systems in their entirety. Its

Design example

The second example involves developing a while loop for the following specification:

$f(x,y) = (x>100 \rightarrow x,y:=x,x+1) \mid (x,y:=x,y)$

To develop this program from its specifications, use the four-step process based on rules D.I through D.V (explained in the main text). First determine if f is realizable by a while loop.

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

Since f is defined for all input, domain(f) includes all values of x and y. Range(f) is some subset of (x,y), so condition D.I is true.

D.II. For $(x,y) \in$ range(f), do we have an identity function, that is, $f(x,y) = (x,y)$?

There are two cases for x: $x>100$ and $x \leq 100$. For the case of $x>100$, we have from the specification that $(x,x+1) \in$ range(f), and $f(x,x+1) = (x,x+1)$, which is an identity. For the case where $x \leq 100$, we know from the specification that $f(x,y) = (x,y)$.

D.III. Find [b] that evaluates to true in domain(f) - range(f) and false in range(f).

Find a predicate b that is false on its range and true elsewhere. Since we want y to take on the value $x+1$ or x to be less than or equal to 100 on the range of f, we know that $(y=x+1)$ OR $(x \leq 100)$ will be true on the range and hence false on domain(f) - range(f). So the negative of this has our desired property: not$((x \leq 100)$ or $(y=+1)) = (x>100)$ and $(y<>x+1)$. Since the loop will exit when this predicate is false, $b = (x>100)$ and $(y<>x+1)$, giving the partial solution

```
while (x>100) and (y<>(x+1)) do
  {d}
```

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

To find a function [d] that preserves the values needed for f, y needs to become $x+1$. So let $d = (x,y := x,x+1)$. Our solution is now

```
while (x>100) and (y<>x+1) do
  {x,y:=x,x+1}
```

or just

```
while (x>100) and (y<>x+1) do
  y:=x+1
```

D.V. Show that the loop must terminate.

We know that b is false on the range of the while statement. Thus, if we can prove that the loop terminates, the current values of x and y when the loop terminates must be a feasible solution.

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 it 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.

The technique presented here was quite manual, with the development of trace tables that grow in complexity as the number of conditionals increases. However, much of the process can be automated. For example, most of the details in a verification proof consist of keeping track of the various trace table executions. But this is a mechanical, syntactic property of programs, and a computer is ideal for carrying out such repetitive tasks. At the University of Maryland, we implemented an extension called FSQ to the Support integrated environment to facilitate such proofs.[8] The goal is to develop a semiautomatic system that guides the user into making the correct decisions. This should greatly ease the problems in developing such proofs.

Program verification — whether using this functional approach or some other approach, like axiomatic or algebraic correctness — is not an easy task. However, programming is not easy, and the need for correct programs is great. Using the functional correctness method described in this article will not guarantee simplicity in developing large correct programs, but it does provide a methodological basis for developing correct programs.

The method described in this article adds to the current set of techniques addressing the important, but extremely difficult, problem domain of program verification. The software engineering field still has a long way to go before program verification becomes an accepted activity in all programming developments. This article simply describes another tool that can be evaluated along with the others in determining the best approach towards good engineering of software. ■

Acknowledgments

I thank Victor Basili and John Gannon for their helpful comments on earlier drafts of this article, and also thank the referees who greatly improved its quality.

Partial support of this work was obtained from Air Force Office of Scientific Research grant 90-0031 to the University of Maryland.

References

1. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming, *Comm. ACM*, Vol. 12, No. 10, Oct. 1969, pp. 576-580, 583.
2. J.V. Guttag and J.J. Horning, "The Algebraic Specifications of Abstract Data Types," *Acta Informatica* Vol. 10, No. 1, Jan. 1978, pp. 27-62.
3. H.D. Mills, "The New Math of Computer Programming," *Comm. ACM*, Vol. 18, No. 1, Jan. 1975, pp. 43-48.
4. J.D. Gannon, R.G. Hamlet, and H.D. Mills, "Theory of Modules," *IEEE Trans. Software Eng.*, Vol. 13, No. 7, July 1987, pp. 820-829.
5. H.D. Mills et al., *Principles of Computer Programming: A Mathematical Approach*, William C. Brown, Dubuque, Iowa, 1987.
6. H.D. Mills et al., "Mathematical Principles for a First Course in Software Engineering," *IEEE Trans. Software Eng.*, Vol. 15, No. 5, May 1989, pp. 550-559.
7. E. Dijkstra, "On the Cruelty of Really Teaching Computer Science," *Comm. ACM*, Vol. 32, No. 12, Dec. 1989, pp. 1,398-1,404.
8. M.V. Zelkowitz, "Evolution Towards a Specifications Environment: Experiences with Syntax Editors," *Information and Software Technology*, Vol. 32, No. 3, Apr. 1990, pp. 191-198.

Marvin V. Zelkowitz is a professor of computer science at the University of Maryland, College Park, with appointments in the Department of Computer Science and the Institute for Advanced Computer Studies. His research interests are in software engineering, programming environments, measurement, and compiler design.

A member of ACM and a senior member of IEEE, he is a past chair of ACM SIGSoft and of the Computer Society's Technical Committee on Software Engineering. He obtained a BS in mathematics from Rensselaer Polytechnic Institute and MS and PhD degrees in computer science from Cornell University.