Formal Verification of Programs

SEI Curriculum Module SEI-CM-20-1.0

December 1988

Alfs T. Berztiss
University of Pittsburgh

Mark A. Ardis
Software Engineering Institute

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.

Draft For Public Review

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Education Program is developing a wide range of materials to support software engineering education. A **curriculum module** identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in *designing* a course. A **support materials** package includes materials helpful in *teaching* a course. Other materials under development include model curricula, textbooks, educational software, and a variety of reports and proceedings.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

SEI curriculum modules may be copied or incorporated into other materials, but not for profit, provided that appropriate credit is given to the SEI and to the original author of the materials.

Comments on SEI educational publications, reports concerning their use, and requests for additional information should be addressed to the Director of Education, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

Comments on this curriculum module may also be directed to the module authors.

Alfs T. Berztiss
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Mark A. Ardis
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Formal Verification of Programs

Acknowledgements

We would like to thank Susan Gerhart, David Gries, Jan Storbank Pedersen, Mary Shaw, and Jeanette Wing for their many helpful comments and suggestions.

Contents

Formal Verification of Programs

Module Revision History

Version 1.0 (December 1988) Draft for public review

Formal Verification of Programs

Capsule Description

This module introduces formal verification of programs. It deals primarily with proofs of sequential programs, but also with consistency proofs for data types and deduction of particular behaviors of programs from their specifications. Two approaches are considered: verification after implementation that a program is consistent with its specification, and parallel development of a program and its specification. An assessment of formal verification is provided.

Philosophy

Central to the development of a program is a *requirements statement*, which tells in precise terms what the program is to accomplish. Parts of this document can be expressed formally, *i.e.*, in a language that has formally defined syntax and semantics. Other parts cannot be so expressed. Our concern here is with the formal component only. This we shall call the *specification* of the program, consistent with the usage adopted in [Berztiss87].

Verification demonstrates that a program is consistent with its requirements. Verification can be either formal or informal. All programs have to be verified, but formal verification is not often practicable.

Nevertheless, formal verification is highly relevant to practical software engineering. First, concern with formal verification increases our understanding of the nature of the program under development. In particular, it causes the software engineer to focus on the precision, consistency, and completeness of the requirements statement. Second, concern with proofs causes the software engineer to opt for the simplest implementation consistent with requirements. This reduces the potential for introducing errors and contributes to the maintainability of the program. Third, it is essential to verify formally some critical parts of software systems that have to be safe, such as life-support systems.

Objectives

The main thrust of this module is the actual use of verification techniques. However, formal verification can be studied from different viewpoints. The programming methodologist is primarily interested in the integration of programming with formal verification. A logician's main concern is the inference system in which the verification takes place. An objective of artificial intelligence is to provide improved heuristics for program verifiers (which are instances of mechanical theorem provers).

A software engineer does the actual formal verification, or requests that it be done. This requires at least the ability to determine its feasibility, which, in turn, requires an understanding of the viewpoints listed above. One purpose of this module is to facilitate this understanding.

A student who has absorbed the material of this module can be expected to:

- Understand the nature of formal verification and its role in the software development process.
- Be familiar with a number of different approaches to formal verification.
- Be able to carry out formal verifications using these approaches.

Prerequisite Knowledge

For a full appreciation of the topics of this module, the student should have had experience both with programming-in-the-small, where program proofs can be practicable, and with programming-in-the-large, where the proof techniques have not yet been developed to a high degree of practicability. In particular, there must be an understanding of the specification process [Rombach87] and of formal specifications [Berztiss87].

Ability to understand formal specifications implies that the student already has some knowledge of predicate logic. Additional topics of predicate logic are introduced in the module itself. The student should also have a general understanding of discrete mathematics at least equivalent to that provided by a three–credit-hour college course. The need here is for mathematical maturity rather than specific course content.

Finally, the student needs to have some understanding of verification and validation in general, an area outlined in [Collofello88].

Module Content

Outline

I. Introduction
  1. Requirements statements and specifications
  2. The meaning of a proof
  3. The meaning of correctness
    a. Correctness: programs
    b. Correctness: data types

II. Predicate Logic
  1. Basic concepts
    a. Notation
    b. Interpretations
    c. Theories
  2. Inference systems
    a. Natural deduction
    b. Resolution
    c. Heuristics for resolution
    d. Induction principles

III. Verification Methods
  1. Hoare logic
    a. The ***while*** language
    b. Extensions and limitations
  2. Dijkstra's approach
    a. Weakest preconditions
    b. Development of programs and proofs in parallel
  3. Mills's functional correctness

IV. Special Problems
  1. Loops
    a. Invariants
    b. Functions
  2. Procedures, parameters, and **goto**'s
  3. Arrays, records, and pointers
  4. Data types
    a. Data abstraction
    b. Consistency and sufficient completeness
    c. Implementation

V. Automation of Verification

VI. Assessment of Verification

Annotated Outline

I. Introduction

1. Requirements statements and specifications

The term "program" can describe any computerized software system or any component of such a system. Thus, the term can be applied to a range of software products: a module that encapsulates a data type, a procedure that formats a string, a database management system, a distributed operating system, a chess player, etc. In most cases, program development obeys a common characteristic: it starts with a requirements statement or a specification.

A requirements statement is a *testable* statement describing properties that a program is to possess. A statement is testable if there is a reasonable expectation that it can be experimentally shown that a program does or does not satisfy the statement. For example, "the program has an average response time of 500 msec" is testable, and so is "the average time between program breakdowns is at least 8 hours, with 95% confidence." "The program has good response times" is too vague to be testable.

A specification is a *provable* statement describing properties that a program is to possess. A statement is provable if there is a reasonable expectation that it can be established by a formal logical proof or an informal mathematical argument that a program is or is not consistent with the statement. For example, two provable statements are: "the program satisfies the output predicate $P(x)$," and "for n input items, the running time of the program is at most $O(n^2)$." The statement, "the program has an average response time of 500 msec," may be provable; but "the average time between program breakdowns is at least 8 hours with 95% confidence" is not provable. Provable statements are also testable.

The qualifier "reasonable expectation" relates to both theoretical and practical problems. In general, it is undecidable whether a program will stop. Moreover, even in decidable cases, the proof of a program may be too difficult to find. Also, the statement "the average time between program breakdowns is at least 8 years, with 95% confidence" is testable in principle, but not in practice.

This module deals primarily with formal verification of the functional properties of simple programs, *i.e.*, with proofs that show that the outputs generated by these programs are consistent with the specifications of the programs. The application of formal techniques to specifications themselves must also be

considered. In a specification, one should aim at consistency and completeness. For example, if a sorted array $C[1..n]$ is to be produced from array $A[1..n]$, then the statement that $C[i+1] \geq C[i]$ is to hold for all i in $1..n-1$ is an incomplete specification. For completeness, it has also to be stated that C is a permutation of A. The specification would be inconsistent if it further required that $C[1]$ contain the largest element of C. In the specification of programs, completeness cannot be formally proven, and consistency is difficult to prove; but both of these properties can be formally proven for specifications of data types.

2. The meaning of a proof

In mathematics, a proof is the derivation of a statement from a set of theorems, and the statement itself then becomes a theorem. In a formal proof, every step of the proof is the application of an inference rule of logic. In an informal proof, instead of such strict justification, an appeal is made to the mathematical knowledge of the reader. This distinction is discussed in [Culik83].

Computer scientists are interested in formal proofs because they can be generated by computer, or, at least, a computer scientist and a computer can cooperate interactively in the generation of a formal proof. In any case, a formal proof can be checked mechanically; the checking of an informal proof is not as easy. Moreover, when a program and its proof are developed side by side, the programmer gains a better understanding of both the program and the programming process in general. Software engineers are interested in formal proofs because the underlying proof methodologies cause them to consider the precision, consistency, and completeness of specifications, the clarity of implementations, and the consistency of implementations and specifications. This, in turn, results in more reliable software, even when an explicit formal proof is not performed.

3. The meaning of correctness

a. Correctness: programs

In the context of computer programs, the proof of a program is the demonstration that it is consistent with its specification. For a formal proof, the program is regarded as a formal object, *i.e.*, as a string in a language for which there is a formal syntax and a formal semantics. Similarly, the specification has to be written in a formal language. Two levels of proof are distinguished: *total* proof of a program shows that the program is consistent with its specification; *partial* or *conditional* proof shows that it is consistent with the specification *provided execution of the program terminates*. Verification is surveyed in [Berg82]. [Gehani86] is primarily a collection of papers on specification, but some of the papers are relevant to verification as well.

Sometimes it is argued that a functional program (one written in Lisp, say) or a logical program (*e.g.*, one written in Prolog) is its own specification and is therefore correct by definition [Turner85, Henderson86, Kowalski85]. However, it can also be argued that a specification and an implementation represent two points of view of a problem, which can be useful in understanding or debugging both. For example, testing may reveal errors in either the specification or the code [Gannon81]. Taking this a step further, [Hoare87] argues for setting up and proving consistent two or more independent specifications for a software system; it appears that the concept of institutions [Goguen86] would be useful for this.

b. Correctness: data types

For abstract data types, correctness arises in two contexts. First, the specification is correct if it is consistent and complete. (These terms will be discussed later.) Second, an implementation of the abstract data type is correct if it can be shown to be consistent with the specification.

II. Predicate Logic

1. Basic concepts

a. Notation

A *formal proof* establishes that a statement is a logical consequence of axioms or previously proven theorems. This requires a framework for reasoning, which is provided by the language of *logic*. Syntactic rules define which combinations of the symbols of this language are *well-formed formulas* (WFFs) and which WFFs are *clauses*. Note that every WFF can be expressed as a set of clauses. Depending on how restrictive or permissive the syntax, we obtain different logics: propositional logic, first-order predicate logic with equality, second-order predicate logic, etc.

b. Interpretations

A WFF is *interpreted* by assigning a meaning to every constant symbol and free variable in it. An interpreted WFF is a *statement*. Different interpretations of the same WFF transform it into different statements. Semantic rules of the logic permit a statement to be evaluated, where the evaluation of a statement yields the value *true* or the value *false*. A WFF is *valid* if and only if it evaluates to *true* for every interpretation; it is *satisfiable* if and only if it evaluates to *true* for at least one interpretation. It is *nonvalid* if and only if it evaluates to *false* for at least one interpretation; and it is *unsatisfiable* if and only if it evaluates to *false* for every interpretation.

An interpretation in which a WFF evaluates to *true* is a *model* for this WFF. The notion of a model can be extended to any arbitrary subset Q of the set of WFFs: an interpretation is a model of Q if every WFF in Q evaluates to *true* under this interpretation. A WFF w, which need not belong to Q, is a *logical consequence* of Q if it evaluates to *true* under every interpretation of Q.

A *calculus* is a set of axiomatic schemas and a set of inference rules that are to generate WFFs with a particular property. The calculus is *sound* if all the WFFs generated by the calculus have the property, and *complete* if it can generate all the WFFs with this property. First-order predicate calculus is sound and complete.

The property of greatest interest is validity: a *predicate calculus* is to construct only valid WFFs. A predicate logic is *decidable* if an algorithm exists that determines, for every WFF of the logic, whether or not it is valid. It is *semi-decidable* if an algorithm exists that determines for every valid WFF that it is valid, but the algorithm need not halt if the WFF is nonvalid. It is *undecidable* if it is not even *semi-decidable*. Second-order predicate logic is undecidable. First-order predicate logic is semi-decidable (which follows from the soundness and completeness of the calculus); with some syntactic restrictions, it can become decidable.

c. Theories

Given the set of WFFs of first-order predicate logic, a subset Q of this set of WFFs is a *theory* if it has a model and if all logical consequences of Q are also in Q. The existence of a model implies consistency. The closure property implies that a theory contains at least all the valid WFFs of predicate logic, but it also contains additional formulas that relate to a particular application, such as a theory of strings.

A good introduction to this material can be obtained by reading both Section 2.1 of [Manna74] and Chapter 2 of [Loeckx84]). For examples of interpretations, see [Boolos80]. The better known data types of computer science are studied as theories in [Manna85].

2. Inference systems

a. Natural deduction

The natural deduction system is so called because it is supposed to embody the principles that humans use in day-to-day reasoning. The main component of the system is a set of pairs of rules, one pair for each logical operator. One of the rules permits the introduction of the operator, the other its elimination. A thorough treatment can be found in [Manna74]. [Gries81] gives a simplified introduction. Here is a simple version of the *or-elimination* rule:

$$\frac{a \rightarrow c,\ b \rightarrow c,\ a \vee b}{c}$$

The notation asserts that if the expression (or expressions) above the line is (are all) true, then the expression (or expressions) below the line is (are) also true. Here we have the assertion: "If a implies c, and b implies c, and a or b is true, then also c is true." Some proofs are simpler when equivalence transformations are used instead of the rules of the natural deduction system (see Chapter 2 of [Gries81]).

b. Resolution

Suppose we have to prove that, on assuming certain WFFs to hold, Q also holds. One approach is to take the assumptions and the negation of Q, namely $\neg Q$, convert these WFFs to a set of clauses [Manna74], and eliminate clauses from this set until nothing is left, *i.e.*, until the *null clause* has been obtained. The null clause stands for contradiction, which means that $\neg Q$ does *not* hold, and that therefore Q *does* hold. Commonly, the clauses are eliminated by application of the *resolution principle*, which, in a very simple form, is the inference rule

$$\frac{P \vee Q,\ \neg P \vee R}{Q \vee R}$$

The precise form of the general rule is, however, rather complicated [Manna74, Genesereth87], primarily because the clauses targeted for elimination have to be made "compatible," which is done by *unification*.

c. Heuristics for resolution

If the ultimate null clause is obtained by applications of the resolution principle alone, the process is called *resolution refutation*. The economy of having to deal with just one inference rule has led to a widespread use of resolution refutation in automatic theorem provers. The heuristics that make resolution refutation practicable are surveyed in [Genesereth87].

d. Induction principles

Program proofs, particularly of recursive programs and of programs based on recursively defined data types, often use induction. Here, however, the common induction principle, which ranges over natural numbers, has to be generalized. *Noetherian* or *structural* induction is one such generalization. This and other induction principles are defined in [Manna74] and [Loeckx84].

III. Verification Methods

1. Hoare logic

a. The ***while*** language

In 1969, Hoare [Hoare69] introduced an axiomatic approach to program correctness. He used a very simple language that has only assignment, sequencing of statements, **if-then-else,** and **while** (hence, the ***while*** *language*). Each of these constructs is interpreted by a proof rule. The assignment rule is an axiom, and the other three are inference rules. This interpretation is the semantics of the language and is used in program proofs. A fourth inference rule had to be added just for program proofs. All the rules use some form of the expression $\{P\}S\{R\}$, which reads "if statement P is true before execution of S, then, provided S terminates, R will be true after execution of S." If this expression is true, then P is a *precondition* of S, and R is a *postcondition*.

Two examples of the rules are the assignment axiom and the composition rule. The assignment axiom is

$$\{R[t/x]\}\, x := t\, \{R\}$$

where *R[t/x]* is a version of R in which every occurrence of x is replaced by the expression t. The axiom may be read as "if $R[t/x]$) is true before execution of $x := t$, then R is true afterwards." The composition rule:

$$\frac{\{P\}\, S_1\, \{R\},\ \{R\}\, S_2 \{Q\}}{\{P\}\ S_1; S_2\ \{Q\}}$$

may be read as "given a composition of two statements such that R is a postcondition of the first statement with respect to precondition P and a precondition of the second statement with respect to postcondition Q, then P is a precondition and Q a postcondition of the composition."

The nature of the proof of a program S is as follows. The programmer supplies statements P and R, which supposedly describe the intended purpose of the program. Statement P defines properties of the input of the program; R defines properties of the output. Expression $\{P\}S\{R\}$ is then the hypothesis to be proven. Let us write the program as $S_1; S_2; \ldots ; S_n$. Then, a postcondition of S_1 is derived from precondition P of the entire program. The composition rule permits us to use this postcondition for the precondition of $S_2; \ldots ; S_n$, and so forth, until R is shown to be a postcondition of the entire program by showing it to be a postcondition of S_n.

b. Extensions and limitations

This approach has been extended to other constructs of programming languages, and the extensions have been surveyed from a logician's point of view in [Apt81]. The approach has been applied to the specification of "real" programming languages, notably Pascal [Hoare73].

However, there are several limitations. An important one has to do with the data types on which the program operates. Apt's survey contains the proof of a program for division by repeated subtraction. Despite the extreme simplicity of the program, appeal has to be made twice to the properties of integers. This means that an automated program verifier has to be provided with knowledge about properties of different data types, which is a difficult task in general, but one which has been successfully handled for integers and arrays [Constable82]. More seriously, it may not be possible to express the postcondition of a program in first-order logic. This is the case for a program that computes the transitive closure of a relation. On the theoretical plane, it has been demonstrated that there are programming languages for which a Hoare logic does not exist [Clarke79], as well as languages for which Hoare-type proofs are exponential in length [Cherniavsky79]. Moreover, the proof of a program in Hoare logic guarantees correctness of implementation only if the program is processed by a compiler based on the same Hoare logic.

2. Dijkstra's approach

a. Weakest preconditions

Suppose we have shown that $\{P\}S\{R\}$ is true when P states that the input is any positive integer. If we now were to suspect that program S is applicable to negative integers as well, we would have to repeat the proof with a modified P. Dijkstra noted that the most important statement is R, and that we should not be required to supply P; instead, the program proof should define the set of *all* input values for which the output satisfies R [Dijkstra75].

For S and R as before, define a predicate $wp(S, R)$. This, the *weakest precondition* of S with respect to R, defines the set of all inputs for which S produces *in finite time* an output that satisfies R. Note that in Dijkstra's usage, $\{P\}S\{R\}$ reads "if statement P is true before execution of S, then S *terminates* with R being true." Under this usage, $\{P\}S\{R\}$ and $P \rightarrow wp(S, R)$ are equivalent, *i.e.*, the set of inputs defined by P is a subset of the set of inputs defined by $wp(S, R)$. Ideally, we should always determine the weakest precondition of a program with respect to a given result assertion R. In real life, we may not be clever enough to do so; then the implication $P \rightarrow wp(S, R)$ still allows us to apply Dijkstra's approach with a less than perfect determination of the input range.

b. Development of programs and proofs in parallel

Weakest preconditions are often associated with the development of a program and its proof in parallel, with the proof ideas guiding program development. [Dijkstra76] is a monograph on this approach. Good texts also exist, based on Dijkstra's language [Gries81] and Hoare's language [Backhouse86]. See [Dromey88] for a set of heuristics that can help in systematic program development.

Dijkstra's language differs from Hoare's, but assignment and composition should have the same meaning in both. We have, as expected:

$$wp(\text{“}x := t\text{”}, R) = R[t/x]$$

$$wp(\text{“}S_1; S_2\text{”}, R) = wp(S_1, wp(S_2, R))$$

Note that a *wp* guarantees total correctness. The significant difference between the two approaches is in the way they deal with looping, because the *wp*-approach has to ensure that a loop terminates. Note that proofs of programs written in Dijkstra's language may be exponential in length [Jones81], but this is unlikely to arise in practice.

3. Mills's functional correctness

Harlan Mills has proposed a method of specification and proof based on functions and relations rather than preconditions and postconditions. In this method, called the *functional correctness* method, one describes a program or procedure by a function that relates output values of the active variables to their input values. This is the *specification function*. [Mills86] contains an introduction to this method based on Pascal, including heuristics for developing specifications. The theory and context for the method are covered in [Linger79]. Application of this method within IBM has been called the "Cleanroom approach" [Mills87].

For assignment and conditional statements, functional correctness provides simple rules. Assignment to a variable is described by a function from the type of the variable to the same type. Conditional statements are described by conditional expressions in terms of functions. Composition of statements is described by composition of functions. A proof consists of a demonstration that the function computed by a program (or sequence of statements) is equivalent to the specification function. In practice, one often shows that the function computed *contains* the specification function, since specifications are often incomplete. Proof of correctness of implementations is discussed in [Gannon87].

IV. Special Problems

1. Loops

a. Invariants

The proving of programs that contain only simple constructs, such as assignments and conditional statements, is trivial. Proofs of programs that contain loops require a greater degree of creativity. Here the correctness problem is tackled by proposing a *loop invariant*, *i.e.*, a condition that is to hold at every entry to and every exit from the loop. To take a simple example, an invariant for a loop that accumulates the sum of the elements of an array $A[1..n]$ as the value of variable *sum* is $sum = \sum_{i=1}^{k} A[i]$, where k counts the times the loop has been executed. When the loop invariant is known and is closely related to the postcondition of the loop, the verification of the loop is straightforward and can be carried out according to a checklist of five steps given on page 145 of [Gries81]. Two of the steps relate to a bound function: loop termination is shown by proving that the bound function is never negative and that it decreases in each iteration of the loop. For our example, the bound function is *n-k*. (See also [Gries82].)

The postcondition for the summation loop is $sum = \sum_{i=1}^{n} A[i]$, so the loop invariant is closely related to the postcondition, and application of the checklist is easy. Often this is not so, and the formulation of the loop invariant requires much creativity. Automation of the derivation of invariants in simpler instances has been investigated [Dershowitz81]. Gries suggests that loops be constructed from proposed postconditions, and he devotes an entire chapter of [Gries81] to the development of loop invariants. This is essential reading, but it should be noted that a full appreciation of the heuristics proposed by Gries may require the study of more examples than Gries provides. The development of loop invariants from the properties of a problem is studied in [Turski84].

Intermittent assertions have been proposed as an alternative to invariant assertions [Burstall74, Manna78]. Burstall arrives at the intermittent assertions by means of symbolic execution. Symbolic execution forms the basis for a verification method described in [Hantler76] as well. [Manna78] claims that intermittent assertions lead to cleaner proofs of total correctness, but this claim has been questioned [Gries79a]. The method of intermittent assertions is based on temporal logic. For a brief introduction to temporal logic, see [Manna81]. Proofs of the Schorr-Waite marking algorithm have been undertaken using both invariant assertions [Gries79b] and intermittent

assertions [Topor79]. Another method that does not require invariant assertions for loops is subgoal induction [Morris77]; its relation to earlier methods is discussed in [King80] and to functional correctness, in [Dunlop82]. A taxonomy of induction principles can be found in [Cousot82]. [Mili86] surveys the development of correct programs by reasoning about them under both approaches, Dijkstra-Gries and functional correctness.

b. Functions

In Mills's functional correctness, one must guess a function that describes a loop. Then, the function must be shown to be equivalent to an unrolled version of the loop. For example, suppose that we wish to find the function computed by "**while** *B* **do** *S*." We guess a function *f*, and show $f = [\textbf{if } B \textbf{ then } S] \circ f$ where the square brackets denote "the function computed by." Additionally, we must show that the function correctly describes the case when the loop body is never executed. This is done by showing equivalence with the identity function for that special case. Rather than show that a loop terminates, one shows that the domain of the function guessed includes the domain of the actual loop function.

Guessing loop functions is as difficult as guessing loop invariants. Initialized loops are easier to describe because one can use the initialization knowledge to simplify the description. Otherwise, one must describe the function computed by the loop, assuming any number of initial iterations. [Dunlop82] describes this phenomenon and compares functional correctness to Hoare's axiomatic method.

2. Procedures, parameters, and **goto**'s

Procedures provide another source of problems for verification. Good treatments of procedures can be found in [Gries81] and [Martin83]. For procedure calls without parameters, or calls where the arguments match the parameters, the semantics of the call may be described by macro-expansion. This leads to a very simple rule. There are two remaining problems: substitution of parameters and recursion. The first problem is solved by introducing a rule of substitution that prohibits aliasing and side effects. The second problem is solved with induction:

1. Prove that the procedure is correct whenever there are no recursive calls.
2. Assuming that the procedure is correct for *n* recursive calls, show that it is correct for *n*+1 recursive calls.

Procedures are treated similarly in both the weakest precondition and the functional correctness methods.

Goto's and **exit**'s are more difficult to handle. Attempts to deal with these constructs are described in [Clint72, Kowaltowski77]. Part of the difficulty with unrestricted transfers of control is the rapid increase in complexity of the programs that use them. Good programming practice dictates that such transfers be used only in special cases, *e.g.*, exits from loops and error exits. Proof rules for such restricted cases are given in [Arbib79, Luckham80, Cristian84].

3. Arrays, records, and pointers

Hoare's axiom of assignment [Hoare69] assumes that the variable on the left-hand side of the assignment is a simple identifier, and substitution for a simple identifier is easy to define. However, most programming languages allow more complicated left-hand sides of assignments, including array references, record selection, and pointers. Proofs involving assignment to one of these objects require additional machinery. The usual method is to introduce functions that perform the appropriate substitution and selection. The resulting expressions are much larger, but intellectually no more difficult to use, than simple identifiers.

[Hoare73] handles these special cases for Pascal. [Gries81] and [Reynolds81] use a notation that significantly improves the readability of proofs involving arrays. [Luckham79] provides a nice simplification of the semantics of all of these constructs.

4. Data types

a. Data abstraction

Data abstraction has a number of meanings. In the main, specification of data types is based either on an abstract model or an algebraic approach. Under the abstract model approach, the operations of the data type are expressed in terms of operations of a simpler type, *e.g.*, stack operations as operations on sequences. On the other hand, the basis of an algebraic specification is a self-contained set of axioms. For references, see [Berztiss87]. The application of specifications of data types to programming-in-the-large is addressed in [Goguen86], [Guttag85], [Liskov86], and [Wulf81].

Verification, in the context of abstract data types, relates to three activities. First, one should be able to determine whether an implementation that derives from a specification has certain properties before any implementation actually takes place. Second, it should be possible to demonstrate the consistency and completeness of a specification. Third, an implementation should be provably consistent with its specification. Examples of reasoning about properties of prospective implementations from their specifications are found in [Guttag80], [Shaw81], [Chi85], and [Wing87]. Limitations in reasoning about abstract data types

are discussed in [MacQueen85]. Implementation correctness is considered in [Guttag78b].

b. Consistency and sufficient completeness

An algebraic specification defines a theory. Consequently, it is consistent if there is a model. But the mere existence of a model is not enough; the model should express user expectations with regard to the data type being specified. The model is then a "correct" implementation. This approach is discussed in [Guttag78b] and [Goguen78]. Completeness in the logical sense does not leave much implementation freedom, which has led to the introduction of the concept of sufficient completeness [Guttag78a]. This is highly technical material. [Berztiss83] provides an introduction for the nonspecialist.

c. Implementation

In an early paper on the verification of an implementation of a data type with respect to its specification, Hoare describes the relationship between an abstract operation and its implementation in terms of an "abstraction mapping" R [Hoare72]. This mapping from representation to abstraction must obey a commutative property. For example, let an abstract operation f be implemented by a concrete function p and denote its function symbol by P. Then mapping R must obey the relationship: $R(P(x)) = f(R(x))$. Intuitively, the left-hand side of this equation describes the result of finding the abstract value of the result of applying the implemented operation, and the right-hand side describes the result of applying the abstract operation to the appropriate original abstract value.

V. Automation of Verification

Teaching Consideration: *Students often ask whether there is any way to automate some of the more tedious steps of the program verification process. It is a good idea to discuss such tools; the discussion will help to clarify the distinction between those tasks that are simple and mechanical and those that require inspiration or creativity.*

Tools to support the verification of programs usually fall into one of three classes:

1. **Theorem provers.** These tools assist the user in proving theorems, using a large body of knowledge about interesting types, such as integers.
2. **Verification condition generators.** These tools produce logical formulas (verification conditions) from a program and its specification. The user must then prove that those formulas are theorems, in order to show that the program is correct.
3. **Verification systems.** These tools assist the user in developing a proof of a program, using previously proven lemmas (which might have been proven by hand or with the assistance of a theorem prover) and verification conditions.

The first class, theorem provers, includes a great diversity of tools and approaches. Some theorem provers try to do as much as possible before requiring the user's intervention (*e.g.*, Boyer-Moore [Boyer79, Boyer81b]), while others encourage the user to provide assistance or advice (*e.g.*, NuPRL [Constable86]). Verification condition generators are rather simple to construct and are often included in verification systems. Because they embody language-specific knowledge, they are usually compiler-specific. A verification condition generator for FORTRAN is described in [Boyer81a]. Verification systems, such as Affirm [Sunshine82] and Gypsy [Good85], are usually language-dependent (since they often use a built-in verification condition generator) and are very interactive. All of these systems require an extensive learning period before they can be used effectively.

[Lindsay88] is an excellent survey of verification technology. [Cheheyl81] is an earlier survey, restricted in the number of systems covered, but highly informative about the four systems it deals with. The strengths and weaknesses of current verification technology are surveyed in [Levitt85] and [Craigen87].

VI. Assessment of Verification

Criticisms of formal verification have come from various quarters: an argument is advanced in [DeMillo-79] that a proof has to convince its reader, which formal proofs may fail to do; Naur [Naur82] notes some negative effects of basing verification on specifications that are difficult to understand; a distinction between mathematical and logical proofs (or scientific and mathematical theories) is emphasized in [Culik83] and [Turski86]. Either by emphasizing the difference between mathematical and logical proofs or by ignoring this difference, all of these criticisms suggest that closer attention should be given to the separation of the proof of an algorithm from the proof of a program.

The simultaneous development of an algorithm and a program can work very well for simple algorithms [Gries81, Backhouse86]. However, there are some algorithms that are difficult to deal with under any approach to correctness. Iteration schemas of numerical analysis belong to this class. Here the proof of convergence of an iterative algorithm, *i.e., proof of termination* in our terminology, may occupy a numerical analyst several years.

Abstract data types provides an area in which the formal approach has been a success. Routine application of data abstraction and reasoning from abstract specifications is just beginning in industry [Cohen86], about

10-15 years after the initial research, which is the normal time span for a successful innovation to become accepted in practice.

Teaching Considerations

Suggested Schedules

Teaching the material of this module can easily occupy one or two entire semesters, in the unlikely event that so much time is available. In a more realistic setting, the material could be part of a course on verification and validation, say one-third or one-half of the course. The exact length of time available for program proving will depend on the background knowledge of the students, *e.g.*, whether or not they have previously studied predicate logic.

Keeping in mind likely time restrictions, the instructor has to select carefully the material to be covered. Emphasis should be on Section III, where one verification method should be selected for detailed discussion. The other methods should be given in brief overview. Although most of the technical material of Section II would be omitted when time is limited, the instructor should have some familiarity with this background material.

Worked Examples and Exercises

Before teaching this material, the instructor should have carried out as many formal verifications as needed to acquire the self-confidence that is essential for conveying the importance of the material to the students. At the very least, examples from the literature must be studied thoroughly before they are presented in class. Many examples of supposedly formal proofs have been published without the full justification required in a formal proof. In some cases, the proofreading could have been better as well.

When a proof is incremental, *i.e.*, when a program is developed in parallel with its proof, the development should be done in class. It should be done interactively, with full class participation. On the other hand, proofs of existing programs tend to be very boring, and they should *not* be developed in class. The examples should be duplicated for distribution to the class and copied to overhead transparencies for classroom presentation.

Good exercises are still rather hard to come by in this area. Instead of restricting attention to books on verification, one can also look for exercises in introductory textbooks on programming in particular languages. It is essential that the instructor solve exercises before assigning them.

Bibliography

Apt81

Apt, K. R. "Ten Years of Hoare's Logic: A Survey —Part I." *ACM Trans. Prog. Lang. and Syst. 3* (1981), 431-483.

> ***Abstract:*** *A survey of various results concerning Hoare's approach to proving partial and total correctness of programs is presented. Emphasis is placed on the soundness and completeness issues. Various proof systems for while programs, recursive procedures, local variable declarations, and procedures with parameters, together with the corresponding soundness, completeness, and incompleteness results, are discussed.*
>
> A fairly technical survey with an extensive bibliography of 55 entries.

Arbib79

Arbib, M. A., and S. Alagic. "Proof Rules for Gotos." *Acta Informatica 11* (1979), 139-148.

> ***Abstract:*** *We offer a program specification format adapted to statements with multiple exits, and use it to present proof rules to replace the somewhat unsatisfactory treatment of jumps in [Clint72]. We justify the* bridled *use of "goto"s in return exits, failure exits, and loops with jumps in the middle. To exemplify our methodology, we prove the function* lookup.
>
> A well-presented contribution to a rather specialized topic.

Ashcroft76

Ashcroft, E. A., M. Clint, and C. A. R. Hoare. "Remarks on 'Program Proving: Jumps and Functions by M. Clint and C. A. R. Hoare'." *Acta Informatica 6* (1976), 317-318.

> Corrects an error in [Clint72].

Backhouse86

Backhouse, R. C. *Program Construction and Verification*. Englewood Cliffs, N. J.: Prentice-Hall, 1986.

> An introductory text that emphasizes the need to consider program development and program verification in parallel. An excellent source of worked examples that illustrate this approach. Note that Backhouse has been careful to avoid overlap of examples with [Gries81].

Berg82

Berg, H. K., W. E. Boebert, W. R. Franta, and T. G. Moher. *Formal Methods of Program Verification and Specification*. Englewood Cliffs, N. J.: Prentice-Hall, 1982.

> A wide-ranging survey of verification. The bibliography is extensive (166 entries).

Berztiss83

Berztiss, A. T., and S. Thatte. "Specification and Implementation of Abstract Data Types." In *Advances in Computers, Vol. 22*. New York: Academic Press, 1983, 295-353.

> This survey of the equational algebraic specification of abstract data types contains a section on verification of implementations of the abstract data types.

Berztiss87

Berztiss, A. *Formal Specification of Software.* Curriculum Module SEI-CM-8-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

> This module discusses formal specification of software, with emphasis on functional properties.

Boolos80

Boolos, G. S., and R. C. Jeffrey. *Computability and Logic*. Cambridge, England: Cambridge University Press, 1980.

> A very readable introduction to the topics of the title.

Boyer79

Boyer, R. S., and J. S. Moore. *A Computational Logic*. New York: Academic Press, 1979.

> This book defines a logical theory tailored to the needs of reasoning about Lisp programs and describes many techniques that the authors have developed for proving theorems in this theory. Particular emphasis is on induction.

Boyer81a

Boyer, R. S., and J. S. Moore. "A Verification Condition Generator for Fortran." In *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore, eds. London: Academic Press, 1981, 9-101.

Abstract: *This paper provides both a precise specification of a subset of Fortran 66 and Fortran 77 and a specification of the verification condition generator we have implemented for that subset. Our subset includes all the statements of Fortran 66 except the following: Read, Write, Rewind, Backspace, Endfile, Format, Equivalence, Data and Block Data. We place some restrictions on the remaining statements; however, our subset includes certain uses of Common, adjustable array dimensions, function subprograms, subroutine subprograms with side effects, and computed and assigned Go Tos. Unusual features of our system include a syntax checker that enforces all our syntactic restrictions on the language, the thorough analysis of aliasing, the generation of verification conditions to prove termination, and the generation of verification conditions to ensure against such runtime errors as array bound violations and arithmetic overflow. We have used the system to verify several running Fortran programs. We present one such program and discuss its verification.*

A demonstration that some programs written in older languages can still be verified. The example is the Boyer-Moore substring matching algorithm.

Boyer81b

Boyer, R. S., and J. S. Moore. "Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures." In *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore, eds. London: Academic Press, 1981, 103-184.

Abstract: *We describe a sound method for permitting the user of a mechanical theorem-proving system to add executable code to the system, thereby implementing a new proof strategy and modifying the behavior of the system. The new code is mechanically derived from a function definition conceived by the user but proved correct by the system before the new code is added. We present a simple formal method for stating within the theory of the system the correctness of such functions. The method avoids the complexity of embedding the rules of inference of the logic in the logic. Instead, we define a meaning function that maps from objects denoting expressions to the values of those expressions under a given assignment. We demonstrate that if the statement of correctness for a given "metafunction" is proved, then the code derived from that function's definition can be used as a new proof procedure. We explain how we have implemented the technique so that the actual application of a metafunction is as efficient as hand-coded procedures in the implementation language. We prove the correctness of our implementation. We discuss a useful metafunction that our system has proved correct and now uses routinely. We discuss the main obstacle to the introduction of metafunctions: proving them correct by machine.*

An extension of [Boyer79].

Burstall74

Burstall, R. M. "Program Proving as Hand Simulation with a Little Induction." In *Proc. IFIP World Congress 1974*. Amsterdam: North-Holland, 1974, 308-312.

Abstract: *A method of proving facts about programs is presented in an informal manner, in the hope that it will have some intuitive appeal to programmers. It derives essentially from Manna's method, but it is influenced by the recent idea of 'executing' a program symbolically as part of the proof process. Some examples are worked out, including one to invert a permutation in situ and one to traverse a tree; the latter seems to come out rather easily this way. Finally this technique and the Floyd one are related to a system of modal logic.*

An interesting feature of Burstall's approach is the use of inductive proofs on iterative programs.

Cheheyl81

Cheheyl, M. H., M. Gasser, G. A. Huff, and J. K. Millen. "Verifying Security." *ACM Computing Surveys 13* (1981), 279-339.

Abstract: *Four automated specification and verification environments are surveyed and compared: HDM, FDM, Gypsy, and AFFIRM. The emphasis of the comparison is on the way these systems could be used to prove security properties of an operating system design.*

The acronyms HDM and FDM stand for Hierarchical Development Methodology and Formal Development Methodology, respectively. HDM is based on the specification language Special, and the (non-interactive) Boyer-Moore theorem prover. FDM makes use of the Ina Jo specification language and an interactive theorem prover. Gypsy is a highly integrated environment intended for the incremental verification of software. In Affirm, software development is regarded as the specification and implementation of abstract data types, and specifications are written as algebraic axioms. Although this survey deals specifically with the verification of security, it provides clear descriptions of the four verification methodologies listed above and is an invaluable guide to further reading. The largest application example known at the time is indicated for each environment; these examples correct the rather negative prognosis for automated verification expressed in [DeMillo79].

Cherniavsky79

Cherniavsky, J. C., and S. N. Kamin. "A Complete and Consistent Hoare Axiomatics for a Simple Programming Language." *J. ACM 26* (1979), 119-128.

> ***Abstract:*** *A simple programming language L_m is defined for which a complete axiomatics is obtainable. Completeness is shown by presenting a relatively complete Hoare axiomatics, demonstrating, by direct construction, that the first-order theory of addition P_+ is expressive, and noting than P_+ is complete. It is then shown that L_m is maximal with this property. Further, a notion of complexity of a Hoare system is introduced based upon the lengths of proofs (disregarding proofs in the underlying logic), and the system $L_m P_+$ is shown to have polynomial complexity. The notion is shown to be nontrivial by presenting a language for which any Hoare axiom system has exponential complexity.*
>
> This very theoretical paper shows that proofs in a Hoare logic can be exponential in length.

Chi85

Chi, U. H. "Formal Specification of User Interfaces: A Comparison and Evaluation of Four Axiomatic Approaches." *IEEE Trans. Software Eng. SE-11* (1985), 671-685.

> ***Abstract:*** *Few examples of formal specification of the semantics of user interfaces exist in the literature. This paper presents a comparison of four axiomatic approaches which we have applied to the specification of a commercial user interface—the line editor for the Tandy PC-1 Pocket Computer. These techniques are shown to result in complete and relatively concise descriptions. A number of useful and nontrivial properties of the interface are formally deduced from one of the specifications. In addition, a direct implementation of the interface is constructed from a formal specification. Limitations of these specification examples are discussed along with future research work.*
>
> The significance of this paper to formal verification lies in the full proofs of three statements about the user interface that are given in the appendix.

Clarke79

Clarke, E. M. "Programming Language Constructs for Which It Is Impossible to Obtain Good Hoare Axiom Systems." *J. ACM 26* (1979), 129-147.

> ***Abstract:*** *Hoare axiom systems for establishing partial correctness of programs may fail to be complete because of (a) incompleteness of the assertion language relative to the underlying interpretation or (b) inability of the assertion language to express the invariants of loops. Cook has shown that if there is a complete proof system for the assertion language (*i.e. *all true formulas of the assertion language) and if the assertion language satisfies a natural expressibility condition then a sound and complete axiom system for a large subset of Algol may be devised. We exhibit programming language constructs for which it is impossible to obtain sound and complete sets of Hoare axioms even in this special sense of Cook's. These constructs include (i) recursive procedures with procedure parameters in a programming language which uses static scope of identifiers and (ii) coroutines in a language which allows parameterless recursive procedures. Modification of these constructs for which sound and complete systems of axioms may be obtained are also discussed.*
>
> The results of this paper have minor practical relevance to the verification of programs, but they indicate theoretical limitations of the Hoare approach to program correctness.

Clint72

Clint, M., and C. A. R. Hoare. "Program Proving: Jumps and Functions." *Acta Informatica 1* (1972), 214-224. See [Ashcroft76] for a correction.

> ***Abstract:*** *Proof methods adequate for a wide range of computer programs have been expounded in [Hoare69] and [Hoare71]. This paper develops a method suitable for programs containing functions, and a certain kind of jump. The method is illustrated by the proof of a useful and efficient program for table lookup by logarithmic search.*
>
> The ability to deal with programs that contain functions is essential for modular software development. The techniques of this paper are therefore very important for software verification by means of the Hoare approach.

Cohen86

Cohen, B., W. T. Harwood, and M. I. Jackson. *The Specification of Complex Systems*. Wokingham, England: Addison-Wesley, 1986.

> A fairly short introduction (143 pages) that explores some aspects of the electronic office by means of equational algebraic specification and the Vienna Development Method (VDM). Specification of concurrent systems is briefly touched on as well.

Collofello88

Collofello, J. S. *Introduction to Software Verification and Validation.* Curriculum Module SEI-CM-13-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

> The author provides the following capsule description: "This module provides a framework for understanding the curriculum modules in the verifi-

cation and validation area. Verification and validation techniques are introduced and their applicability discussed. Approaches to integrating these techniques into comprehensive verification and validation plans are also addressed."

Constable82

Constable, R. L., S. D. Johnson, and C. D. Eichenlaub. *An Introduction to the PL/CV2 Programming Logic*. Berlin: Springer-Verlag, 1982. Springer-Verlag Lecture Notes in Computer Science, No. 135.

This paper describes a formal system for reasoning about integers, arrays, and programming language commands in the PL/CS language. The reasoning can be checked by the PL/CV Proof Checker.

Constable86

Constable, R. L., *et al*. *Implementing Mathematics with the NuPRL Proof Development System*. Englewood Cliffs, N. J.: Prentice-Hall, 1986.

The NuPRL ("New Pearl") system is an environment for deriving proofs in constructive type theory. PRL systems are concerned with three kinds of objects: problems, their solutions, and explanations of the solutions. When a problem is expressed as a formula in a logical theory and a solution of the problem as some computable function, then a proof of the formula can be regarded as an explanation of the solution. However, formal constructive proofs can be executed and, thus, can become solutions. Hence the formalization of an informal explanation becomes the development of a program.

Cousot82

Cousot, P., and R. Cousot. "Induction Principles for Proving Invariance Properties of Programs." In *Tools and Notions for Program Construction*, D. Neel, ed. Cambridge, England: Cambridge University Press, 1982, 75-119.

Abstract: *We propose sixteen sound and complete induction principles for proving program invariance properties. We study their relationships and show that they can be derived from each other by commuting mathematical transformations. Only five of these induction principles correspond to already known invariance proof methods. We choose a non-conventional induction principle and construct corresponding partial correctness, nontermination and clean behavior proof methods. When constructing these new proof methods, we informally apply our mathematical approach published earlier. This essentially consists in decomposing the global inductive invariant involved in the induction principle into an equivalent set of local invariants and in deriving the corresponding verification condition.*

The proof methods based on the induction principles are all applied to a single example, division by repeated subtraction. This is an attempt to bring order into a rather confusing field. Primarily of interest to the specialist.

Craigen87

Craigen, D. "Strengths and Weaknesses of Program Verification Systems." In *Proc. 1st European Software Engineering Conf.*, H. K. Nichols and D. Simpson, eds. Berlin: Springer-Verlag, 1987, 396-404. Springer-Verlag Lecture Notes in Mathematics, No. 289.

Abstract: *For over a decade, major research efforts have been directed at developing and applying Program Verification Systems. Particular examples are the Gypsy Verification Environment (at The University of Texas at Austin and Computational Logic, Inc.), and Affirm-85 (at General Electric, Schenectady, New York).*

In this paper, I discuss the putative strengths and weaknesses of the current generation of verification systems, describe the characteristics of a system which can be developed at low technical risk, and then describe briefly a research effort, at I.P. Sharp Associates, to develop a new verification system called EVES.

The weaknesses referred to in the title are, primarily: logical unsoundness permitted by the systems, expense of use, excessive dependence on low-level inference rules, the scarcity of reusable mathematical theories, specification languages with low expressive power, and poor environmental support. The strengths, which do not fully exist in current systems, are: the verification system as guarantor of logical soundness, as processor of trivial detail in proofs, and as tracker of re-verification needs when specifications change.

Cristian84

Cristian, F. "Correct and Robust Programs." *IEEE Trans. Software Eng. SE-10* (1984), 163-174.

Abstract: *The design of programs which are both correct and robust is investigated. It is argued that the notion of an exception is a valuable tool for structuring the specification, design, verification, and modification of such programs. The syntax and semantics of a language with procedures and exception handling are presented. A deductive system is proposed for proving total correctness and robustness properties of programs written in this language.*

The system is both sound and complete. It supports proof modularization, in that it allows one to reason

separately about fault-free and fault-tolerant system properties. Since the programming language considered closely resembles CLU or Ada, the presented deductive system is easily adaptable for verifying total correctness and robustness properties of programs written in these, or similar, languages.

The main advantage of Cristian's approach over similar approaches is that the effect of faults, which are detected as occurrences of exceptions, can be treated separately from the verification of fault-free program properties.

Culik83

Culik, K., and M. M. Rizki. "Logic versus Mathematics in Computer Science Education." *ACM SIGCSE Bulletin 15*, 1 (Feb. 1983), 14-20.

Abstract: *Informal mathematical proofs admit and require interpretation while formal logic proofs suppress (abstract from) meanings. The former is closely related to problem solving and computer programming. The latter, which is commonly used for proving program correctness, complicates this procedure because it separates problem solving from programming. A constructive mathematical proof in finite discrete mathematics of an existential theorem is a computer program if the pertinent data structures and functions are expressed in a programming language. Several detailed examples of graph theoretical problems and theorems are presented along with their constructive proofs and corresponding programs.*

One can gain important insights regarding programs and their proofs from this paper, without necessarily agreeing with all of the authors' conclusions.

DeMillo79

DeMillo, R. A., R. J. Lipton, and A. J. Perlis. "Social Processes and Proofs of Theorems and Programs." *Comm. ACM 22* (1979), 271-280.

Abstract: *It is argued that formal verification of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage. It is felt that ease of formal verification should not dominate program language design.*

This controversial article argues that program verification is impracticable except, perhaps, in some instances. Although these instances are not as few as the authors imply (see [Cheheyl81], for example), some of the arguments of the article are valid, and it should be required reading for every computer scientist.

Dershowitz81

Dershowitz, N., and Z. Manna. "Inference Rules for Program Annotation." *IEEE Trans. Software Eng. SE-7* (1981), 207-222.

Abstract: *Methods are presented whereby an Algol-like program given together with its specifications can be documented automatically. The program is incrementally annotated with invariant relations that hold between program variables at intermediate points in the program text and explain the actual workings of a program regardless of whether it is correct. Thus, this documentation can be used for proving correctness of programs or may serve as an aid in debugging incorrect programs.*

The annotation techniques are formulated as Hoare-like inference rules that derive invariants from the assignment statements, from the control structure of the program, or, heuristically, from suggested invariants. The application of these rules is demonstrated by examples that have run on an experimental implementation.

The abstract conveys both the content and the intent of this important paper very well. The references give excellent coverage of earlier work.

Dijkstra75

Dijkstra, E. W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." *Comm. ACM 18* (1975), 453-457.

Abstract: *So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be shown.*

This paper introduced weakest preconditions.

Dijkstra76

Dijkstra, E. W. *A Discipline of Programming*. Englewood Cliffs, N. J.: Prentice-Hall, 1976.

Weakest preconditions expounded by the master himself. The book suffers from the absence of an index and a bibliography.

Dromey88

Dromey, R. G. "Systematic Program Development." *IEEE Trans. Software Eng. SE-14* (1988), 12-29.

Abstract: *A constructive method of program development is presented. It is based on a simple strat-*

egy for problem decomposition that is claimed to be more supportive of goal-oriented programming than the Wirth-Dijkstra top-down refinement method. With the proposed method, a program is developed by making a sequence of refinements, each of which can establish the postcondition for a corresponding sequence of progressively weaker preconditions until a mechanism has been composed that will establish the postcondition for the original given precondition for the problem. The strategy can minimize case analysis, simplify constructive program proofs, and ensure a correspondence between program structure and data structure.

Often the postcondition for a program is given in a form that makes it difficult to construct a program that will satisfy this postcondition. This paper deals with transformations of the postcondition that make this task easier.

Dunlop82

Dunlop, D. D., and V. R. Basili. "A Comparative Analysis of Functional Correctness." *ACM Computing Surveys 14* (1982), 229-244.

Abstract: *The functional correctness technique is presented and discussed. It is also explained that the underlying theory has an implication for the derivation of loop invariants. The functional verification conditions concerning program loops are then shown to be a specialization of the commonly used inductive assertion verification conditions. Next, the functional technique is compared and contrasted with subgoal induction. Finally, the difficulty of proving initialized loop programs is examined in light of both the inductive assertion and functional correctness theories.*

A brief overview of functional correctness with a discussion of its theoretical foundations.

Gannon81

Gannon, J., P. McMullin, and R. Hamlet. "Data-Abstraction Implementation, Specification, and Testing." *ACM Trans. Prog. Lang. and Syst. 3* (1981), 211-223.

Abstract: *A compiler-based system DAISTS that combines a data-abstraction implementation language (derived from the Simula class) with specification by algebraic axioms is described. The compiler, presented with two independent syntactic objects in the axioms and implementation code, compiles a "program" that consists of the former as test driver for the latter. Data points, in the form of expressions using the abstract functions and constant values, are fed to this program to determine if the implementation and axioms agree. Along the way, structural testing measures can be applied to both code and axioms to evaluate the test data. Although a successful test does not conclusively demonstrate the consistency of axioms and code, in practice the tests are seldom successful, revealing errors. The advantage over conventional programming systems is threefold:*

(1) The presence of the axioms eliminates the need for a test oracle; only inputs need be supplied.

(2) Testing is automated: a user writes axioms, implementations, and test points; the system writes the test drivers.

(3) The results of tests are often surprising and helpful because it is difficult to get away with "trivial" tests: what is not significant for the code is liable to be a severe test of the axioms, and vice versa.

The reader should note that the testing of implementations of data abstractions against their algebraic specifications can reveal errors in either.

Gannon87

Gannon, J. D., R. G. Hamlet, and H. D. Mills. "Theory of Modules." *IEEE Trans. Software Eng. SE-13* (1987), 820-829.

Abstract: *Because large-scale software development is a struggle against internal program complexity, the modules into which programs are divided play a central role in software engineering. A module encapsulating a data type allows the programmer to ignore both the details of its operation, and its value representations. It is a primary strength of program proving that as modules divide a program, making it easier to understand, so do they divide its proof. Each module can be verified in isolation, then its internal details ignored in a proof of its use. This paper describes proofs of module abstraction based on functional semantics, and contrasts this with the Alphard formalism based on Hoare logic.*

In this, one of a series of papers on functional correctness, modules are identified as data types. First a formal semantics of modules is developed; this is then used to formulate a theory for proving module implementations correct with respect to their specifications. The approach is illustrated with an example, the data type of rational numbers. The Alphard formalism is described in [Shaw81].

Gehani86

Gehani, N., and A. D. McGettrick, eds. *Software Specification Techniques*. Wokingham, England: Addison-Wesley, 1986.

A collection of 21 papers, most of which have contributed significantly to shaping the field of software specification and verification.

Genesereth87

Genesereth, M. R., and N. J. Nilsson. *Logical Foundations of Artificial Intelligence*. Los Altos, Calif.: Morgan Kaufmann, 1987.

Chapters 3-5 of this book give a good introduction to logic, particularly to uses of the resolution principle.

Goguen78

Goguen, J. A., J. W. Thatcher, and E. R. Wagner. "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types." In *Current Trends in Programming Methodology, Vol. 4*, R. T. Yeh, ed. Englewood Cliffs, N. J.: Prentice-Hall, 1978, 80-149.

Abstract: *Abstract data types have been claimed a powerful tool in programming (as in Simula and CLU), both from the viewpoint of user convenience and that of software reliability. Recently algebra has emerged as a promising method for the specification of abstract data types; this makes it possible to prove the correctness of implementations of abstract types. It also raises the question of the correctness of the specifications and the proper method for handling run-time errors in abstract types. Unfortunately not all the algebra underlying these issues is entirely trivial, nor has it been adequately developed or explained. In this chapter we show how a reasonable notation for many-sorted algebras makes them just as manageable as one-sorted (universal) algebras, and we present comparatively simple yet completely rigorous statements of the major algebraic issues underlying abstract data types. We present a number of specifications, with correctness proofs for some; the issue of error messages is thoroughly explained, and the issue of implementations is broached.*

This is an important contribution to the literature on abstract data types, but its authors assume familiarity with some fairly advanced mathematics.

Goguen86

Goguen, J. A. "One, None, A Hundred Thousand Specification Languages." In *Proc. IFIP World Congress 1986*. Amsterdam: North-Holland, 1986, 995-1003.

Abstract: *Many different languages have been proposed for specification, verification, and design in computer science; moreover, these languages are based upon many different logical systems. In an attempt to comprehend this diversity, the theory of institutions formalizes the intuitive notion of a "logical system." A number of general linguistic features have been defined "institutionally" and are available for any language based upon a suitable institution. These features include generic modules, module hierarchies, "data constraints" (for data abstraction) and multiplex institutions (for combining multiple logical systems). In addition, institution morphisms support the transfer of results (as well as associated artifacts, such as theorem provers) from one language to another. More generally, institutions are intended to support as much computer science as possible independently of the underlying logical system.*

This viewpoint extends from specification languages to programming languages, where, in addition to the programming-in-the-large features mentioned above, it provides a precise basis for a "wide spectrum" integration of programming and specifications. A logical programming language is one whose statements are sentences in an institution, whose operational semantics is based upon deduction in that institution, giving a "closed world" for a program. This notion encompasses a number of modern programming paradigms, including functional, logic, and object-oriented, and has been useful in unifying these paradigms, by unifying their underlying institutions, as well as in providing them with sophisticated facilities for data abstraction and programming-in-the-large.

This wide-ranging but somewhat technical paper contains useful references to earlier work on institutions.

Good85

Good, D. "Mechanical Proofs About Computer Programs." In *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, eds. Englewood Cliffs, N. J.: Prentice-Hall, 1985, 55-75.

Abstract: *The Gypsy verification environment is a large computer program that supports the development of software systems and formal, mathematical proofs about their behavior. The environment provides conventional development tools, such as a parser for the Gypsy language, an editor and a compiler. These are used to evolve a library of components that define both the software and precise specifications about its desired behaviour. The environment also has a verification condition generator that automatically transforms a software component and its specification into logical formulas that are sufficient to prove that the component always runs according to specification. Facilities for constructing formal, mechanical proofs of these formulas are also provided. Many of these proofs are completed automatically without human intervention. The capabilities of the Gypsy system and the results of its application are discussed.*

An outline of the Gypsy system. In addition to the conventional tools of text editor, parser, pretty-printer, interpreter, compiler, and Ada translator, the

system also contains a verification condition generator, a simplifier, a proof checker, and an optimizer. The verification condition generator produces logical conditions that are sufficient to verify that an implementation meets its specification, and the optimizer produces logical conditions that guarantee the validity of various program optimizations. The simplifier reduces the complexity of logical expressions, and the proof checker performs truth-preserving transformations of logical expressions. The proof checker is interactive.

Gries79a

Gries, D. "Is Sometime Ever Better Than Always?" *ACM Trans. Prog. Lang. and Syst. 1* (1979), 258-265.

Abstract: *The "intermittent assertion" method for proving programs correct is explained and compared with the conventional methods. Simple conventional proofs of iterative algorithms that compute recursively defined functions, including Ackermann's function, are given.*

Gries shows by means of examples of program proofs that the conventional approach can be as effective as that of intermittent assertions.

Gries79b

Gries, D. "The Schorr-Waite Graph Marking Algorithm." *Acta Informatica 11* (1979), 223-232.

Abstract: *An explanation is given of the Schorr-Waite algorithm for marking all nodes of a directed graph that are reachable from one given node, using the axiomatic method.*

The weakest precondition approach is used to demonstrate the correctness of the Schorr-Waite algorithm, which is a method for traversing a structure without the use of a stack. This is a companion paper to [Topor79].

Gries81

Gries, D. *The Science of Programming*. New York: Springer-Verlag, 1981.

The first part of this book is an excellent source of material on logic. Gries believes that a program and its specification, in the form of assertions, should be developed side-by-side. This can work very well for programming-in-the-small.

Gries82

Gries, D. "A Note on a Standard Strategy for Developing Loop Invariants and Loops." *Science of Computer Programming 2* (1982), 207-214.

Abstract: *A by-now-standard strategy for developing a loop invariant and loop was developed in [Dijkstra76] and explained in [Gries81]. Nevertheless, its use still poses problems for some. The purpose of this paper is to provide further explanation. Two problems are solved that, without this further explanation, seem difficult.*

The two problems are the determination of a non-empty sequence of adjacent array elements whose sum is a minimum, and the finding of the largest square of true elements in a rectangular Boolean matrix.

Guttag78a

Guttag, J. V., and J. J. Horning. "The Algebraic Specification of Abstract Data Types." *Acta Informatica 10* (1978), 27-52.

Abstract: *There have been many recent proposals for embedding abstract data types in programming languages. In order to reason about programs using abstract data types, it is desirable to specify their properties at an abstract level, independent of any particular implementation. This paper presents an algebraic technique for such specifications, develops some of the formal properties of the technique, and shows that these provide useful guidelines for the construction of adequate specifications.*

This paper introduces the concept of sufficient completeness of an algebraic specification of a data type.

Guttag78b

Guttag, J. V., E. Horowitz, and D. R. Musser. "Abstract Data Types and Software Validation." *Comm. ACM 21* (1978), 1048-1064.

Abstract: *A data abstraction can be naturally specified using algebraic axioms. The virtue of these axioms is that they permit a representation-independent formal specification of a data type. An example is given which shows how to employ algebraic axioms at successive levels of implementation. The major thrust of the paper is twofold. First, it is shown how the use of algebraic axiomatizations can simplify the process of proving the correctness of an implementation of an abstract data type. Second, semi-automatic tools are described which can be used both to automate such proofs of correctness and to derive an immediate implementation from the axioms. This implementation allows for limited testing of programs at design time, before a conventional implementation is accomplished.*

The abstract conveys the content of this important paper very well.

Guttag80

Guttag, J., and J. J. Horning. "Formal Specification as a Design Tool." In *Software Specification Techniques*, N. Gehani and A. D. McGettrick, eds. Reading, Mass.: Addison-Wesley, 1986, 187-208.

This paper advocates the use of formal specifications and develops a formal specification of a display system. Section 4 is relevant here. It deals with the formalization of questions such as "If I add the same component to the contents of two pictures that appear to be identical, will the resulting pictures also appear to be identical?" and with the formal derivation of answers to the questions from the specifications. Although not program proving, this is a formal approach to software validation.

Guttag85

Guttag, J. V., J. J. Horning, and J. M. Wing. "The Larch Family of Specification Languages." *IEEE Software 2*, 5 (Sept. 1985), 24-36.

Abstract: *Larch specifications are two-tiered. Each one has a component written in an algebraic language and another tailored to a programming language.*

The two-tiered approach permits Larch to bring together abstract data types and programming in-the-large.

Hantler76

Hantler, S. L., and J. C. King. "An Introduction to Proving the Correctness of Programs." *ACM Computing Surveys 8* (1976), 331-353.

Abstract: *This paper explains, in an introductory fashion, the method of specifying the correct behavior of a program by the use of input/output assertions and describes one method for showing that the program is correct with respect to those assertions. An initial assertion characterizes conditions expected to be true upon entry to the program and a final assertion characterizes conditions expected to be true upon exit from the program. When a program contains no branches, a technique known as symbolic execution can be used to show that the truth of the initial assertion upon entry guarantees the truth of the final assertion upon exit. More generally, for a program with branches, one can define a symbolic execution tree. If there is an upper bound on the number of times each loop in such a program may be executed, a proof of correctness can be given by a simple traversal of the (finite) symbolic execution tree.*

However, for most programs, no fixed bound on the number of times each loop is executed exists and the corresponding symbolic execution tree is infinite. In order to prove the correctness of such programs, a more general assertion structure must be provided. The symbolic execution tree of such programs must be traversed inductively rather than explicitly. This leads naturally to the use of additional assertions which are called "inductive assertions."

Earlier approaches to symbolic execution put this technique somewhere between testing and formal verification. Here, however, the combination of symbolic execution with the use of inductive assertions results in a true variant of formal verification.

Henderson86

Henderson, P. "Functional Programming, Formal Specifications, and Rapid Prototyping." *IEEE Trans. Software Eng. SE-12* (1986), 241-250.

Abstract: *Functional programming has enormous potential for reducing the high cost of software development. Because of the simple mathematical basis of functional programming it is easier to design correct programs in a purely functional style than in a traditional imperative style. We argue here that functional programs combine the clarity required for the formal specification of software design with the ability to validate the design by execution. As such they are ideal for rapidly prototyping a design as it is developed. We give an example which is larger than those traditionally used to explain functional programming. We use this example to illustrate a method of software design which efficiently and reliably turns an informal description of requirements into an executable formal specification.*

The intent of Henderson's paper is to support the view that functional programs are their own specifications and that functional programming should, therefore, become a primary tool in the software development process.

Hoare69

Hoare, C. A. R. "An Axiomatic Basis for Computer Programming." *Comm. ACM 12* (1969), 576-580, 583.

Abstract: *In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.*

The original paper on Hoare's method. The important theoretical and practical advantages alluded to in the abstract have indeed followed from a pursuance of the topics of this paper.

Hoare71

Hoare, C. A. R. "Procedures and Parameters—An Axiomatic Approach." In *Symposium on Semantics of Algorithmic Languages*, E. Engeler, ed. Berlin: Springer-Verlag, 1971, 102-116. Springer-Verlag Lecture Notes in Mathematics, No. 188.

The first paper to extend the method of [Hoare69] to include procedures. Better treatments of this topic are to be found in [Martin83] and [Gries81].

Hoare72

Hoare, C. A. R. "Proof of Correctness of Data Representations." *Acta Informatica 1* (1972), 271-281.

Abstract: *A powerful method of simplifying the proofs of program correctness is suggested; and some new light is shed on the problem of functions with side-effects.*

Here Hoare considers correctness of data representations, *i.e.*, demonstrations that a concrete representation exhibits all the properties expected of it by an "abstract program."

Hoare73

Hoare, C. A. R. and N. Wirth. "An Axiomatic Definition of the Programming Language Pascal." *Acta Informatica 2* (1973), 335-355.

Abstract: *The axiomatic definition method proposed in [Hoare69] is extended and applied to define the meaning of the programming language Pascal. The whole language is covered with the exception of real arithmetic and goto statements.*

This paper showed that the axiomatic approach does not have to be confined to "toy" programming languages.

Hoare87

Hoare, C. A. R. "An Overview of Some Formal Methods for Program Design." *Computer 20*, 9 (Sept. 1987), 85-91.

Abstract: *The design of a small program, like that of a large system, requires a variety of formal methods and notations, related by mathematical reasoning.*

Hoare advocates two formal specifications for software of critical importance, which are to be shown consistent or equivalent by means of a proof.

Jones81

Jones, N. D., and S. S. Muchnick. "Complexity of Flow Analysis, Inductive Assertion Synthesis and a Language Due to Dijkstra." In *Program Flow Analysis*, S. S. Muchnick and N. D. Jones, eds. Englewood Cliffs, N. J.: Prentice-Hall, 1981, 380-393.

The authors question the practicability of mechanized verification of arbitrary programs on the basis of their observation that even for very simple programs written in a simple programming language, the lengths of their proofs may not have polynomial bounds.

King80

King, J. C. "Program Correctness: On Inductive Assertion Methods." *IEEE Trans. Software Eng. SE-6* (1980), 465-479.

Abstract: *A study of several of the proof of correctness methods is presented. In particular, the form of induction used is explored in detail. A relational semantic model for programming languages is introduced and its relation to predicate transformers is explored. A rather elementary viewpoint is taken in order to expose, as simply as possible, the basic differences of the methods and the underlying principles involved. These results were obtained by attempting to thoroughly understand the "subgoal induction" method.*

Subgoal induction [Morris77] is compared to earlier approaches.

Kowalski85

Kowalski, R. "The Relation Between Logic Programming and Logic Specification." In *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, eds. Englewood Cliffs, N. J.: Prentice-Hall, 1985, 11-27.

Abstract: *Formal logic is widely accepted as a program specification language in computing science. It is ideally suited to the representation of knowledge and the description of problems without regard to the choice of programming language. Its use as a specification language is compatible not only with conventional programming languages but also with programming languages based entirely on logic itself. In this paper I shall investigate the relation that holds when both programs and program specifications are expressed in formal logic.*

In many cases, when a specification completely defines the relations to be computed, there is no syntactic distinction between specification and program. Moreover the same mechanism that is used to execute logic programs, namely automated deduction, can also be used to execute logic specifi-

cations. The only difference between a complete specification and a program is one of efficiency. A program is more efficient than a specification.

Kowalski argues that a complete logical specification is indistinguishable from a logical program; the only observable difference is one of efficiency.

Kowaltowski77

Kowaltowski, T. "Axiomatic Approach to Side Effects and General Jumps." *Acta Informatica 7* (1977), 357-360.

Abstract: *Hoare's axiomatic method is applied in order to describe two controversial features: side effects and general jumps. The relative simplicity of this description suggests that reasons for the exclusion of these features from programming languages are subtler than it has been thought.*

The simplicity of the description alluded to in the abstract does not imply simplicity of proofs of programs with side effects and general jumps, or even an ability to understand such programs.

Levitt85

Proc. VERkshop III. K. N. Levitt, S. D. Crocker, and D. Craigen, eds. *ACM Software Engineering Notes 10*, 4 (Aug. 1985).

The most recent workshop on formal verification. Papers address the state of current technology, theory, and applications.

Lindsay88

Lindsay, P. A. "A Survey of Mechanical Support for Formal Reasoning." *Software Engineering J. 3* (1988), 3-27.

This survey examines seven support systems in detail and introduces eleven others. The systems discussed in detail: LCF (Logic for Computable Functions), NuPRL, Veritas, Isabelle, Affirm, the Boyer-Moore system, and Gypsy. The bibliography contains 87 items.

Linger79

Linger, R. C., H. D. Mills, and B. I. Witt. *Structured Programming: Theory and Practice*. Reading, Mass.: Addison-Wesley, 1979.

In the functional correctness setting, program correctness is defined as a correspondence between a program and its intended function. This notion is thoroughly explored in Chapter 6 of this book.

Liskov86

Liskov, B., and J. Guttag. *Abstraction and Specification in Program Development*. Cambridge, Mass.: MIT Press, 1986. Distributed by McGraw-Hill.

This textbook on rigorous program development is based on the programming language CLU. Chapter 11 deals with verification.

Loeckx84

Loeckx, J., and K. Sieber. *The Foundations of Program Verification*. Chichester and Stuttgart: Wiley-Teubner, 1984.

This book is strong on the mathematical foundations of program verification and contains a thorough discussion of the semantics of programming languages. It introduces several approaches to verification, including Hoare's axiomatic approach and Milner's LCF; but it has few examples. This limits its usefulness as a textbook. (A second edition, published in 1987, does not seem to have gone beyond the correction of misprints—at least, the bibliography has not been updated.)

Luckham79

Luckham, D. C., and N. Suzuki. "Verification of Array, Record, and Pointer Operations in Pascal." *ACM Trans. Prog. Lang. and Syst. 1* (1979), 226-244.

Abstract: *A practical method is presented for automating in a uniform way the verification of Pascal programs that operate on the standard Pascal data structures Array, Record, and Pointer. New assertion language primitives are introduced for describing computational effects of operations on these data structures. Axioms defining the semantics of the new primitives are given. Proof rules for standard Pascal operations on data structures are then defined using the extended assertion language. An axiomatic rule for the Pascal storage allocation operation, NEW, is also given. These rules have been implemented in the Stanford Pascal program verifier. Examples illustrating the verification of programs which operate on list structures implemented with pointers and records are discussed. These include programs with side effects.*

The examples of this paper deal with side effects in pointer data structures, reachability in linear lists, and the implementation of an event queue.

Luckham80

Luckham, D. C., and W. Polak. "Ada Exception Handling: An Axiomatic Approach." *ACM Trans. Prog. Lang. and Syst. 2* (1980), 225-233.

Abstract: *A method of documenting exception propagation and handling in Ada programs is proposed. Exception propagation declarations are introduced as a new component of Ada specifications, permitting documentation of those exceptions that can be propagated by a subprogram. Exception*

*handlers are documented by entry assertions. Axioms and proof rules for Ada exceptions are given. These rules are simple extensions of previous rules for Pascal and define an axiomatic semantics of Ada exceptions. As a result, Ada programs specified according to the method can be analyzed by formal proof techniques for consistency with their specifications, even if they employ exception propagation and handling to achieve required results (*i.e., *nonerror situations). Example verifications are given.*

By dealing with a rather specialized topic, this article demonstrates further that formal proofs can be applied to programs written in very rich languages.

MacQueen85

MacQueen, D. B., and D. T. Sannella. "Completeness of Proof Systems for Equational Specifications." *IEEE Trans. Software Eng. SE-11* (1985), 454-461.

Abstract: *Contrary to popular belief, equational logic with induction is not complete for initial models of equational specifications. Indeed, under some regimes (the Clear specification language and most other algebraic specification languages) no proof system exists which is complete even with respect to ground equations. A collection of known results is presented along with some new observations.*

The significance of this paper is largely theoretical. However, its results put limits on reasoning from specifications. For example, one may ask whether a program will respond in a particular way to a given input. It may well do so, but there may be no way to prove this from the specifications.

Manna74

Manna, Z. *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.

Although the parts of the book that deal with program verification are by now rather dated, this book remains an excellent introduction to the predicate calculus and to fixpoints in the verification of recursive programs.

Manna78

Manna, Z., and R. Waldinger. "Is 'Sometime' Sometimes Better Than 'Always'? Intermittent Assertions in Proving Program Correctness." *Comm. ACM 21* (1978), 159-172.

Abstract: *This paper explores a technique for proving the correctness and termination of programs simultaneously. This approach, the "intermittent-assertion" method, involves documenting the program with assertions that must be true at some time when control passes through the corresponding point, but that need not be true every time. The method, introduced by Burstall, promises to provide a valuable complement to the more conventional methods.*

The intermittent-assertion method is presented with a number of examples of correctness and termination proofs. Some of these proofs are markedly simpler than their conventional counterparts. On the other hand, it is shown that a proof of correctness or termination by any of the conventional techniques can be rephrased directly as a proof using intermittent assertions. Finally, it is shown how the intermittent-assertion method can be applied to prove the validity of program transformations and the correctness of continuously operating programs.

This is an elaboration of the technique introduced in [Burstall74].

Manna81

Manna, Z., and A. Pnueli. "Verification of Concurrent Programs: The Temporal Framework." In *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore, eds. London: Academic Press, 1981, 215-273.

Abstract: *This is the first in a series of reports describing the application of Temporal Logic to the specification and verification of concurrent programs.*

We first introduce Temporal Logic as a tool for reasoning about sequences of states. Models of concurrent programs based both on transition graphs and on linear-text representations are presented and the notions of concurrent and fair executions are defined.

The general temporal language is then specialized to reason about those execution states and execution sequences that are fair computations of concurrent programs. Subsequently, the language is used to describe properties of concurrent programs.

The set of interesting properties is classified into Invariance (Safety), Eventuality (Liveness) and Precedence (Until) properties. Among the properties studied are: Partial Correctness, Global Invariance, Clean Behavior, Mutual Exclusion, Deadlock Absence, Termination, Total Correctness, Intermittent Assertions, Accessibility, Starvation Freedom, Responsiveness, Safe Liveness, Absence of Unsolicited Response, Fair Responsiveness and Precedence.

In the following reports of this series we use the temporal formalism to develop proof methodologies for proving the properties discussed here.

The earlier parts of this paper provide a useful introduction to temporal logic and to the issues of func-

tional programming.

Manna85

Manna, Z., and R. Waldinger. *The Logical Basis for Computer Programming, Vol. 1: Deductive Reasoning*. Reading, Mass.: Addison-Wesley, 1985.

A thorough exploration of some basic data types—such as non-negative integers, trees, lists, and sets—as mathematical theories, and reasoning about them.

Martin83

Martin, A. J. "A General Proof Rule for Procedures in Predicate Transformer Semantics." *Acta Informatica 20* (1983), 301-313.

Abstract: *A proof rule for the procedure call is derived for procedures with value, result and value-result parameters. It is extended to procedures with unrestricted global variables and to recursive procedures. Like D. Gries's proof rule, it is based on the substitution rule for assignment. However, it is more general and much simpler to apply. Assume that* {U} S {V} *has been proved about the procedure body* S. *The proof rule for determining whether a call establishes predicate* E *is based on finding an "adaptation"* A *satisfying* $A \wedge V \Rightarrow E'$, *where* E' *is derived from* E *by some substitution of parameters.*

Mili86

Mili, A., J. Desharnais, and J. R. Gagne. "Formal Methods of Stepwise Refinement of Programs." *ACM Computing Surveys 18* (1986), 231-276.

Abstract: *Of the many ways to express program specifications, three of the most common are: as a pair of assertions, an input assertion and an output assertion; as a function mapping legal inputs to correct outputs; or as a relation containing the input/output pairs that are considered correct. The construction of programs consists of mapping a potentially complex specification into a program by recursively decomposing complex specifications into simpler ones. We show how this decomposition takes place in all three modes of specification and draw some conclusions on the nature of programming.*

The authors classify program design according to the specification mode on which it is based. For each of the three modes listed in the abstract, the authors provide a design system consisting of an assignment rule, decomposition rules, and a generalization rule (systems A, F, and R, respectively). They then use the rules to derive programs for the longest recurring subsequence problem (System A), sorting (System F), and right justification of text (System R). This paper contributes much to a systematization of the rather confusing field of the development of correct programs by reasoning about them during their construction. The authors are rather pessimistic on the practicability of rigorous program development in-the-large and on fully automatic programming in general. A very important paper.

Mills86

Mills, H. D., V. R. Basili, J. D. Gannon, and R. G. Hamlet. *Principles of Computer Programming, A Mathematical Approach*. Newton, Mass.: Allyn and Bacon, 1986.

This book deals with functional correctness as it would be covered in an introductory computer science course.

Mills87

Mills, H. D., M. Dyer, and R. C. Linger. "Cleanroom Software Engineering." *IEEE Software 4*, 5 (September 1987), 19-25.

Abstract: *Software quality can be engineered under statistical quality control and delivered with better quality. The Cleanroom process gives management an engineering approach to release reliable products.*

Morris77

Morris, J. H., and B. Wegbreit. "Program Verification by Subgoal Induction." In *Current Trends in Programming Methodology, Vol. II: Program Validation*, R. T. Yeh, ed. Englewood Cliffs, N. J.: Prentice-Hall, 1977, 197-227.

Abstract: *A new proof method, subgoal induction, is presented as an alternative or supplement to the the commonly used inductive assertion method. Its major virtue is that it can often be used to prove a loop's correctness directly from its input-output specification without the use of an invariant. The relation between subgoal induction and other commonly used induction rules is explored and, in particular, it is shown that subgoal induction can be used as a specialized form of computation induction. Finally, a set of sufficient conditions are presented which guarantee that an input-output specification is strong enough for the induction step of a proof by subgoal induction to be valid.*

The reading of this paper should be followed by that of Section 5 of [Dunlop82], where subgoal induction is discussed as a generalization of functional correctness.

Naur82

Naur, P. "Formalization in Program Development." *BIT 22* (1982), 437-453.

Abstract: *The concepts of specification and formalization, as relevant to the development of programs, are introduced and discussed. It is found that certain arguments given for using particular modes of expression in developing and proving programs correct are invalid. As illustration a formalized description of Algol 60 is discussed and found deficient. Emphasis on formalization is shown to have harmful effects on program development, such as neglect of informal precision and simple formalizations. A style of specifications using formalizations only to enhance intuitive understandability is recommended.*

A thoughtful critique of too formal an approach to software development. It is most important to understand that Naur is not opposed to all formalizations but only to those that obscure meaning.

Reynolds81

Reynolds, J. C. *The Craft of Programming*. Englewood Cliffs, N. J.: Prentice-Hall, 1981.

A text on development of correct programs. The section on interval diagrams for arrays is particularly important.

Rombach87

Rombach, H. D. *Software Specification: A Framework.* Curriculum Module SEI-CM-11-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

This module gives an overview of software specification and provides a framework within which to discuss specification of both process and product types.

Shaw81

Shaw, M., ed. *Alphard: Form and Content*. New York: Springer-Verlag, 1981.

The Alphard approach to programming puts very heavy emphasis on abstractions and on the formal verification of implementations of such abstractions. Most of the ten papers in this collection deal with verification issues.

Sunshine82

Sunshine, C. A., *et al*. "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models." *IEEE Trans. Software Eng. SE-8* (1982), 460-489.

Abstract: *It is becoming increasingly important that communication protocols be formally specified and verified. This paper describes a particular approach—the state transition model—using a collection of mechanically supported specification and verification tools incorporated in a running system called Affirm. Although developed for the specification of abstract data types and the verification of their properties, the formalism embodied in Affirm can also express the concepts underlying state transition machines. Such models easily express most of the events occurring in protocol systems, including those of the users, their agent processes, and the communication channels. The paper reviews the basic concepts of state transition models and the Affirm formalism and methodology and describes their union. A detailed example, the alternating bit protocol, illustrates various properties of interest for specification and verification. Other examples explored using this formalism are briefly described and the accumulated experience is discussed.*

An excellent introduction to Affirm, and a demonstration of its practical utility in a realistic problem domain.

Topor79

Topor, R. W. "The Correctness of the Schorr-Waite List Marking Algorithm." *Acta Informatica 11* (1979), 211-221.

Abstract: *This paper presents a relatively simple proof of a nontrivial algorithm for marking the nodes of a list structure. The proof separates properties of the algorithm from properties of the data on which it operates and is a significant application of the method of "intermittent assertions."*

The correctness of the Schorr-Waite algorithm, which is a method for traversing a structure without the use of a stack, is demonstrated using the intermittent assertion method of [Burstall74] and [Manna78]. This is a companion paper to [Gries79b].

Turner85

Turner, D. A. "Functional Programs as Executable Specifications." In *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, eds. Englewood Cliffs, N. J.: Prentice-Hall, 1985, 29-54.

Abstract: *To write specifications we need to be able to define the data domains in which we are interested, such as numbers, lists, trees and graphs. We also need to be able to define functions over these domains. It is desirable that the notation should be higher order, so that function spaces can themselves be treated as data domains. Finally, given the potential for confusion in specifications involving a large number of data types, it is a practical necessity that there should be a simple syntactic discipline that ensures that only well typed applications of functions can occur.*

A functional programming language with these

properties is presented and its use as a specification tool is demonstrated on a series of examples. Although such a notation lacks the power of some imaginable specification languages (for example, in not allowing existential quantifiers), it has the advantage that specifications written in it are always executable. The strengths and weaknesses of this approach are discussed, and also the prospects for the use of purely functional languages in production programming.

An exposition of the functional language Miranda, in which recursion equations are combined with some notation from set theory. The set notation allows Miranda to be regarded as a language for both specification and programming.

Turski84

Turski, W. M. "On Programming by Iterations." *IEEE Trans. Software Eng. SE-10* (1984), 175-178.

Abstract: *Iterative computations are considered in this paper as a general problem-solving technique. The loop invariant is derived from problem properties rather than from program properties (as is usual in programming literature). To this end, the notion of equisolution states—a special subset of space-state in which lie the iterated trajectories—is introduced.*

The approach described in the abstract is applied to sorting.

Turski86

Turski, W. M. "And No Philosopher's Stone, Either." In *Proc. IFIP World Congress 1986*. Amsterdam: North-Holland, 1986, 1077-1080.

Turski argues that software consists of a formal component to which formal logical verification applies, and of non-formal domain descriptions or scientific theories that have to be experimentally validated (tested).

Wing87

Wing, J. M. "A Larch Specification of the Library Problem." In *Proc. 4th Intl. Workshop on Software Specification and Design*, M. T. Harandi, ed. Silver Spring, Md.: IEEE Computer Society Press, 1987, 34-41.

Abstract: *A claim made by many in the formal specification community is that forcing precision in the early stages of program development can greatly clarify the understanding of a client's problem requirements. We help justify this claim via an example by first walking through a Larch specification of Kemmerer's library problem and then discussing the questions that arose in our process of formalization. Following this process helped reveal mistakes, premature design decisions, ambiguities, and incompleteness in the informal requirements. We also discuss how Larch's two-tiered specification method influenced our modifications to and extrapolations from the requirements.*

Wing gives several examples of reasoning about the properties of a system on the basis of its formal specification.

Wulf81

Wulf, W. A., M. Shaw, P. N. Hilfinger, and L. Flon. *Fundamental Structures of Computer Science*. Reading, Mass.: Addison-Wesley, 1981.

This text introduces some of the basic concepts of computer science. Sections on verification are distributed throughout the text; Chapters 5, 11, and 17 are particularly relevant.