A Theory of Program Correctness, and Algorithms for Proofs

by J. R. Gabriel, R. O. Chapman, and J. L. Kljaich
Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

--- DISCLAIMER ---

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Printed in the United States of America
Available from
National Technical Information Service
U. S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

NTIS price codes
Printed copy: A02
Microfiche copy: A01
A THEORY OF PROGRAM CORRECTNESS, AND ALGORITHMS FOR PROOFS

J. R. Gabriel, R. O. Chapman,* and J. L. Kijaich**

Mathematics and Computer Science Division

January 1987

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

*Permanent address: Wadham College, Oxford OX1 3PN, U.K.
**Permanent address: Bell Laboratories, Naperville-Wheaton Rd., Naperville, IL 60566
A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.
A Theory of Program Correctness, and Algorithms for Proofs

J. R. Gabriel, R. O. Chapman, and J. L. Kijaich

ABSTRACT

A model of program correctness is given where a problem domain is defined by its language, variable names, and an abstract machine defining the semantics of the language. The set of all computations in this domain is shown to be a semigroup. A corresponding statement is true of a more general programming language. A program \( P \) in the general language is an element of the semigroup. If \( P \) performs computations in some domain, a connection can be established between \( P \) and the semigroup of computations in that domain. Methods already used in proofs about hardware are shown to be useful in proofs about this software model. The paradigm is capable of reasoning about multiprocessor hardware and of "proving" theorems about execution times, that is, "performance."

1. Introduction


Here we describe a formal theory of program verification, implementable by the Barrow-Gabriel-Chapman PROLOG algorithm (henceforth abbreviated to BGC).

2. The Formal Theory

2.1. The Model of a Problem Domain

If \( L \) is a programming language, it has an associated abstract machine \( A \), defining its model of computation [Waite 1973]. For our purposes, \( A \) is assumed to be finite-state machine except in a few cases such as models of real analysis, as distinct from computation.

If \( S \) is a set of declarations for variables in \( L \) (implicit or explicit), augmented by the hidden variables of \( A \) (e.g., the ALU condition codes, the return address stack, the program counter), then the operations of \( A \) applied to \( S \) combine associatively and generate a semigroup \( G(L,S,A) \) of all possible sequences of operations of \( A \). A subset of these comprises syntactically correct computations in \( L \). When \( A \) is finite, these computations are finite in size and in number. Further, since \( A \) is finite, the halting problem is decidable [Gabriel and Roberts 1984]; syntactically valid halting computations are computable mappings of \( S \) onto itself and, from a subset of \( G(L,S,A) \), also a semigroup. Henceforth, \( G(L,S,A) \) will refer to this subset. In the case where \( A \) is not a finite-state machine (e.g., in real analysis), the semigroup may be discrete or continuous and infinite.
Given a programming language Lambda, a set Sigma of variables declared in Lambda, and an abstract finite state machine $M$ defining the semantics of Lambda, there is therefore a semigroup $G'(\text{Lambda, Sigma, } M)$, defining valid computations in Lambda.

If the elements of $M$ and Sigma are in 1:1 correspondence with attributes of a real finite-state machine $M_1$, and the correspondence between $M$ and $M_1$ is isomorphism (i.e., $M$ and $M_1$ "work in the same way"), then $M_1$ implements Lambda for Sigma and Lambda programs $M_1$ in Sigma.

2.2. A Definition of Program Correctness

Suppose $G(\text{L,SA})$ is given and it completely describes all valid computations in some problem domain, but no other computations. Suppose also we are given a programming language Lambda and $G'(\text{Lambda, Sigma, } M)$ whose elements are all computations written in Lambda. Consider a computation $P$ in $G'(\text{Lambda, Sigma, } M)$. Because Sigma contains all possible states of $P$, it contains all possible starting states, and therefore all inputs. Suppose that among these state variables are all possible phrases (products of generators) in the semigroup $G(\text{L,SA})$ defining a problem domain, and moreover there is a 1:1 correspondence between the state space $S$ and a subspace $S'$ of Sigma. Let $P$ accept any phrase $I$ (the input of $P$) from $G(\text{L,SA})$ in the sense that $P$ halts after performing a computation. Then if $P$ performs the same mapping of $S'$ (the subspace of Sigma onto itself corresponding to $S$ in $G(\text{L,SA})$ as $I$ does in $S$ of $G(\text{L,SA})$, $P$ is said to correctly implement the specification $G(\text{L,SA})$.

3. Summary of the Fundamental Concept

A programming language $L$, an abstract machine $A$ defining the semantics of $L$, and a set of variables $S$ define a set of phrases in $L$ where, roughly speaking, opcodes from $L$ are verbs and objects from $S$ are nouns. Each phrase in this set is an element of a semigroup $G(\text{L,SA})$ containing all possible computations definable by $L$ upon the variables in $S$.

Given another language Lambda, a set of variables Sigma, and a machine $M$, there is a corresponding semigroup $G'(\text{Lambda, Sigma, } M)$. If a program $P$ in $G'$ accepts all phrases $I$ (inputs) in $G(\text{L,SA})$ and terminates, having transformed a subspace $S'$ of Sigma just as $I$ transforms $S$ in $G(\text{L,SA})$, then $P$ is a "correct" implementation, in Lambda, on the machine $M$, of the computations defined by $G(\text{L,SA})$. Since $A$ is usually, and $M$ always, a finite-state machine, the termination of $P$ is decidable.

4. Descriptions of Programs and Finite-State Automata

In a problem domain whose computations are described by a semigroup $G(\text{L,SA})$, the (usually) finite-state machine $A$ is the computational mechanism, and the language $L$ is a selector determining which of the possible computations should be done. The present work takes advantage of the fact that for any "program" (i.e., software), there exists a corresponding "finite state automaton" implemented purely in hardware. Thus not only does Barrow's work describe finite-state automata $A$ if they are implemented in hardware, but it may be extended to describe automata implemented by programs, and to describe programs themselves.

This paper adapts Barrow's work to describe software. The adaptation is possible only because of the existence of a hardware automaton corresponding to any recursively computable function (i.e., implementable program).

4.1. The BGC System Description Language

A system is described as a module composed of interconnected, simpler modules. A module has input/output variables declared by "port" statements, and internal state variables
declared by "stateDef" statements. The outputs depend on inputs and state variables; state variables in turn depend on input variables and other state variables. These dependencies are given by "stateEqn" statements and "outputEqn" statements. Connections between ports are defined by "connected" statements. Modules being part of a larger system are declared in "part" statements. A module acquires state variables from its parts; the relation between a state variable of a system and a state variable of a part is declared by a "stateMap" declaration.

The BGC algorithm derives stateDef, stateEqn, and outputEqn statements for a module from those for its component modules. If a component has no outputEqn, one is determined by recursion down to subsystems whose behavior is known. The system description language is given in detail by Barrow [1983]; in this report, we provide only examples.

Because the BCG algorithm determines behavior of finite-state machines, and a program $P$ in $G'(\Lambda', \Sigma, M)$ has a description as a finite-state machine, the BCG algorithm can determine the behavior of $P$ as a set of input/state/output mappings, given the behavior of $M$. The main question is how to do this in a manageable way for large programs.

**A Simple Example in PROLOG**

Consider the following PROLOG program:

```prolog
f(X,Y):-
    g1(X,Z),
    g2(Z,Y).
```

The predicates g1 and g2 are invoked with their first arguments instantiated, and return instances of their second arguments according to

```prolog
g1(X,Z):- Z = g1_Func(X).
g2(Z,Y):- Y = g2_Func(Z).
```

where g1_Func and g2_Func are well-defined, recursively computable functions. The BGC definitions for this system follow.

/********************
* x, y, and z correspond to the variables X, Y, Z of the program.
* ********************/

```prolog
module(g1).
port(g1, x(G1), input, var).
port(g1, z(G1), output, var).
outputEqn(g1, z(G1) := g1_Func(x(G1))).

module(g2).
port(g2, z(G2), input, var).
port(g2, y(G2), output, var).
outputEqn(g2, y(G2) = g2_Func(y(G2))).
```
module(f).
part(f, g1(F), g1).
part(f, g2(F), g2).
port(f, x(F), input, var).
port(f, y(F), output, Var).
connected(f, x(F), x(g1(F))).
connected(f, z(g1(F)), z(g2(F))).
connected(f, y(g2(F)), y(F)).

When presented with this input, the BGC algorithm will infer

\[
\text{outputEqn}(f, y(F) := g2\_Func(g1\_Func(x(F))).
\]

Clearly a compiler from PROLOG to BGC input is needed to prove that the PROLOG program as given implements the input/output relation. We expect that either we or another interested party will write this compiler in due course, and prove the necessary properties of the BGC to recursive function translator [Gabriel and Chapman 1986].

One point needs mentioning in connection with proofs of properties. \(G(L,S,A)\) may have some particular recursively computable function in the definition of \(A\), and the BGC algorithm may derive an apparently different expression for corresponding behavior of \(G'(\Lambda,\Sigma,M)\). In this case the expressions must be proved equivalent by rewriting one to the other, using essentially the same rules as those for lambda calculus, together with any additional rules appropriate to the opcodes of \(M\). This matter is discussed by Bundy [1983], Wegner [1963], Church and Rosser [1936], Knuth and Bendix [1967], and others. The essential point is that not all rewriting rules lead to a finite-decision procedure. For example, there are such confluent rules for combinational logic expressed in terms of XOR and AND, but not for NOT and AND. A rewriting algorithm in PROLOG is given by Gabriel [1986]; it is somewhat more powerful than necessary for confluent rewriting, being in fact a Markov string transformation algorithm with stopping rules and some additional heuristics.

4.2. The Representation of Real Arithmetic

Because floating-point arithmetic is used to represent real arithmetic in many computations, a more detailed look at the relation between the two processes as a case of the formalism is both a useful example and necessary work. This is an obviously important case where the abstract machine \(A\) of \(G(L,S,A)\) is not a finite-state automaton, and the relation between \(A\) and a finite automaton \(M\) must be established with care. In most other cases, even if the machine \(A\) is Turing equivalent (e.g., Markov transformations with stopping conditions), the relation between \(A\) and \(M\) is straightforward through establishment by \(M\) of upper bounds in time and space on the computations.

4.2.1. Floating-Point Numbers

A "floating-point number" is a triple \([s,e,m]\), where \(s\) is a single bit, and \(e\) and \(m\) are unsigned integers of fixed lengths \(l(e)\) and \(l(m)\). This triple defines a binary rational number \(n(s,e,m)\) as follows:

1. The attribute \(s\) determines the sign by a mapping \(\text{sign}(s)\) with the discrete range \([+1,-1]\).
2. The attribute \(e\) determines a scale factor \(2^e\).
3. The attribute \(m\) determines a rational binary fraction \(p(m)\), less than 1 as outlined below.
The number \( n(s,e,m) = \text{sign}(s) \times b \times (2^e) \times p(m) \), where \( b \) is a fixed-scale factor called the bias, chosen as nearly as possible to make the solution \( x \) of the equation \( x \times n(s,e,m) \) for non-zero \( m \), also a floating-point number.

The fraction \( p(m) \) is determined as follows: Let \( m \) have binary digits \( m_0, m_1, m_2, m_3, \ldots \), with \( m_0 \) the most significant. Then \( p(m) = m_0 \times (1/2) + m_1 \times (1/4) + m_2 \times (1/8) + \ldots \).

Define Lambda to include the arithmetic operations including division with remainder. Define Sigma to include zero and all other floating-point numbers having reciprocals. The details of \( M \) (the abstract machine implementing Lambda) are defined by the particular implementation of floating-point arithmetic chosen by the designer of \( M \). The IEEE standard for floating-point arithmetic is an example of a good implementation.

4.2.2. The Correctness of an Implementation

Consider \( G(L,S,A) \) where \( L \) and \( A \) define real arithmetic, and \( S \) is a closed interval on the real line whose ends are the algebraically largest and smallest rational binary numbers in Sigma for some \( G'(\text{Lambda}, \text{Sigma}, M) \).

A conventional delta/epsilon proof in real analysis for some number in \( S \) is an infinite decision procedure which is a repeated finite phrase in \( G(L,S,A) \) determining a limit point in \( S \). Since \( L \) and Lambda are the same language, if the decision procedure is terminated at some finite delta and epsilon, the truncated non-terminating computation in \( L \) has an image in Lambda. Let \( I \) be the input (in \( S \)) to the computation in \( L \). It has a closest corresponding input \( I' \) in Sigma. Let the computation in \( G'(\text{Lambda}, \text{Sigma}, M) \) which is the image of that in \( G(L,S,A) \) with input \( I' \) be performed, leading to an output \( O' \) in Sigma. \( O' \) has an image \( O'' \) in \( S \). Let the output of a computation is \( S \) (i.e., using real arithmetic) be 0.

We define the error factor \( f \) of a particular computation as follows. Let \( I'' \) be the image in \( S \) of \( I' \) in Sigma. Take epsilon as equal to \( |I - I''| \), and calculate \( d \times \text{epsilon} \). Then if \( |0 - O''| = f \times d \times \text{epsilon} \), \( f \) is the (forward) error factor of the computation in \( G'(\text{Lambda}, \text{Sigma}, M) \).

A backward error can be similarly defined. If the computation in \( G'(\text{Lambda}, \text{Sigma}, M) \) leads to an \( O'' \) just as above, which would have been the result of a computation starting from \( I''' \) in \( G(L,S,A) \), then \( |I''' - I''| \) is the backward error of the computation.

A "correct" program \( P \) in \( G(L,S,A) \) is derived from some delta/epsilon proof in \( G'(\text{Lambda}, \text{Sigma}, M) \) having a forward or backward \( f \) factor less than some agreed-upon standard of correctness for \( P \). Since the definition of correctness here is a mapping from non-terminating computations in \( G(L,S,A) \) to terminating ones in \( G'(\text{Lambda}, \text{Sigma}, M) \), a more precise definition is simply not possible. As a colleague once remarked, "In the world of computable functions, Cauchy sequences can be fraudulent."

5. Representing Program Flow

A program or program specification is a sequence of statements in Lambda or \( L \), and, for finite programs, these correspond 1:1 to some range of integers \([0..N]\). There is also a mapping from these "statement numbers" onto the "program counter" state variable of most real machines \( M \).

This string of statements may be divided into "basic blocks" such that flow of control enters only at the start of a basic block, and may change discontinuously at the end of a basic block. These basic blocks correspond to non-overlapping, possibly contiguous, segments either of the statement number range or, in a real machine, the set of possible addresses in program space. Both real and abstract programs have a "generalized program counter," henceforth called simply the "program counter."
Inside a basic block, the program counter changes by one instruction count at a time; we call this mode "continuous." At the end of a basic block, control may "fall through," or jump to the start of a basic block, including the current block.

5.1. A BGC Representation for Programs Having This Model

Thus there is representation of program flow by non-overlapping segments of \([0..N]\) on a straight line, which are basic blocks. The end of one basic block is connected to the beginning of another by a curved arc.

This model has a BGC representation where the arcs on \([0..N]\) transform program variables besides the program counter PC, and the curved arcs transform PC alone. Projecting away the PC dimension of the state space, the curved arcs simply bind "live" variables across transitions between basic blocks; that is, they play the role of BGC connections, while the segments of \([0..N]\) have module names, ports, outputEqns, stateEqns, and the rest.

In this BGC model, program state variables and BGC model state variables are distinct. All program variables are non-state variables in the BGC model. The only BGC state variables arise from cyclic flows of control, namely, loops both in the programming sense and in the sense of directed graphs. The "static external" variables often considered as "state" data in ALGOL like programs are simply always "live."

5.2. Representation of Iteration

The formal theory of a model of iteration, related to BGC formalism, has been given by Chapman and Gabriel [1986]. Simply put, it defines a model in lambda calculus consisting of a finite sequence of states

\[ s_0, s_1, s_2, \ldots, s_n \]

such that the successor \( s' \) of some state \( s \) is derived by a classical, discrete contact transformation (see, e.g., Gossick [1967])

\[ s' = t(s) \]

and \( s_n \) is the first state of the sequence such that a proposition

\[ \text{exit}(s_n) \]

is true.

In informal terms, then,

\[ \text{exit}(t(t(... n \text{ times } ...(\text{in(Loop)}))...))) \]

is true and

\[ \text{out(Loop)} := t(t(...(t(\text{in(Loop)})......))) . \]

However, the notation \( t(t(... n \text{ times } ...(\text{in(Loop)}))...)) \) is not valid BGC syntax. The point is analogous to the ideas discussed by Colmerauer [1982: 231-51], and an analogous solution can be found. A pair of mutually recursive functions is defined, one of them evaluating a state function of the iteration. If the state function were algebraically eliminated between them, the preceding relation would result.
First we need to define infix operators "?" and ":=" in the BGC notation, analogous to the same operators in "C." That is,

\[
\]

is equivalent to

\[
\text{if}(C) \text{ then } E := E1 \text{ else } E := E2.
\]

The two mutually recursive equations are

\[
\begin{align*}
\text{stateEqn}(\text{loop}, \text{state}(L) := & \text{start}(	ext{state}(L) ? \text{in}(L): \\
& \text{exit}(	ext{state}(L) ? \text{state}(L): \\
& \text{t}(\text{id}(L), \text{state}(L)) \\
).
\end{align*}
\]

\[
\text{outputEqn}(\text{loop}, \text{out}(L) := \text{state}(L)).
\]

There is one other question of great interest in connection with iteration, that of loop invariants. From our viewpoint a loop invariant is a mapping \(\sigma' = f(\sigma)\) of the program space \(\Sigma\) onto itself such that \(f(t(\sigma)) = t(f(\sigma))\). It seems likely that invariants can be found by a variant of unification without the occurs check, as is discussed by Colmerauer [1982]. Having found an invariant, if \(s_0\) is the start state of the loop and \(s_e\) the exit state, at exit we have the identity

\[
(\text{exit}(s_0), f(s_e) = f(s_0)) = \text{true},
\]

where the notation is pseudo-PROLOG with logical conjunction being performed by the "," operator. It seems at least possible that this may often be rewritten to prove desired results about iterations.

5.3. Representation of Subroutine Calls

A recursion similar to that of the iteration representation can be used to maintain a list of recursive stack frames containing return locations (values of the abstract program counter \(pc(S)\), where \(S\) is the machine state and \(pc(S)\) is an argument of a "constant" declaration in BGC).

In particular, if the previous representation of iteration seems unnatural, a mapping into recursive use of the transform \(t(S)\) can be made.

6. The Question of Program Correctness

Given the usual pair, a specification \(G(L,S,A)\) and a program \(P\) being a phrase in \(G'(\Lambda, \Sigma, M)\), which is the program to be verified, we know there is a subspace \(S'\) of \(\Sigma\) with variables corresponding to those in \(S\) according to the input/output user manual for \(P\). The elements of \(S\) are the opcodes of \(A\) and valid associated operands. There is also a mapping from values of operands in \(S\) and values in \(S'\). Often this is one-to-one, but sometimes, as in the case of the representation of real arithmetic on finite machines, it must be many-to-one. For the rest of this section we assume the mapping of values is 1:1.
In cases where it is not, proofs of correctness divide in two parts, one being analogous to the previous discussion of representations for real numbers, and the other being analogous to derivation of properties for a program performing floating-point arithmetic. This is not a severe problem for the theory: it says simply that the mapping from non-computable functions to computable ones should be discussed separately from the question of whether \( P \) implements some computable function. Thus, with no loss of generality, we may assume the mapping is 1:1, and that variables in \( S \) and \( S' \) have been renamed so that alphabetically identical expressions in \( G \) and the subspace \( S' \) of \( G' \) have the same meaning.

As defined, \( S \) does not contain the opcodes of \( A \), these being found in \( L \) of \( G(L,S,A) \). Since \( P \) is a single element of \( G' \), however, the degrees of freedom represented by \( L \) in \( G(L,S,A) \) appear as another subspace \( L' \) of Sigma containing instructions for \( P \) to perform operations of \( A \) on data values of variables in \( S' \). Thus, simply put, inputs to \( P \) are pairs \([Q,D]\), where \( Q \) is an operation from the specification, and \( D \) is a set of data values.

Execution of \( P \) with input \([Q,D]\) maps \([Q,D]\) into \( D' \), another set of values for variables in \( S' \), this time corresponding to output variables in the specification \( G(L,S,A) \). The semantics of \( A \) define a mapping by \( A \) of \([Q,D]\) into some set of values \( D1 \) in \( S \). The program \( P \) is said to be correct if the mapping induced in \( G' \) by \( P \)

\[ [Q,D] \rightarrow D1' \]

is the same as that induced by \( A \) in \( G \), namely,

\[ [Q,D] \rightarrow D1 . \]

That is, \( D1' \) in \( G'(Lambda, Sigma, M) \) is the same as \( D1 \) in \( G(L,S,A) \).

For most programs \( P \), a case analysis by enumerating all \([Q,D], D1, \) and \( D1' \) is out of the question, although this is what a validation test attempts. But as we shall see, the BGC algorithm can be used to derive the mapping

\[ [Q,D] \rightarrow D1' \]

in symbolic form, essentially as an expression in lambda calculus whose functors are opcodes of \( M \), and whose innermost arguments are variables of \( S' \). Program verification is then the proof that this can be rewritten to the corresponding expression in the semantics of \( L \) of \( G(L,S,A) \) defining the opcode \( Q \) of the abstract machine \( A \). The rewriting rules are those of the lambda calculus together with those in the semantics of \( M \) and \( A \). The algorithm for deciding equivalence is given by Knuth and Bendix [1967] and discussed in more detail by Bundy [1983]. A method to automatically develop a finite decision procedure is given by Lusk and Overbeek [1984]. A rewriting algorithm to actually perform the proof given a proper decision procedure is given by Chapman and Gabriel [1986] and Gabriel [1986].

7. Use of Program Flow Graphs

The derivation of the input/output behavior of a program as a practical process instead of a conceptual one now needs to be discussed in more detail. The discussion shows that the BGC algorithm can be used to obtain input/output relations.

The representation where basic blocks are non-overlapping intervals in \([0,N]\) and jumps are curved arcs connecting an exit from one basic block to possible entries of others may represent subroutine calls in different ways. In one, the call statement is simply a basic block and has no explicit connection to the flowgraph for the subroutine except for being annotated.
by the subroutine name and an argument list. In this representation a program is a set of disjoint flowgraphs, one for each subroutine. Alternatively, the call and return mechanism may be explicitly shown, in which case the flowgraph of a proper program has no disjoint subgraphs. These two have no significant difference in content. Henceforth the first will be used because it is more consonant with established principles of software engineering such as data hiding, subroutine libraries, and module reusability.

7.1. Multiprocessor Architectures

The model is not limited to von Neumann machines, and some conveniences of notation are worth noting for multiprocessor architectures. In such systems, the program space (i.e., the basic blocks in \([0..N]\)) has serially reusable parts. In this case the program counter \(pc(S)\) for each processor is represented by a different-colored token placed on the flowgraph. No two tokens may occupy the same serially reusable section at the same time. This rule is enforced by various kinds of arbitrators resolving contention for resources. It is sometimes useful to color the arcs of the flowgraph according to the processors allowed to use them. In this case shared resources are polychromatic, while local resources "owned" by a processor are monochromatic.

7.2. Detailed Representation of Program Flow

In our standard representation where calls to subroutines are represented by annotated single arcs, the program definition contains only annotations comprising subroutine names and argument lists. Consider first a subroutine containing no calls to other subroutines. The BGC algorithm, together with the model of iteration given earlier in this report, allows a lambda calculus expression to be constructed, giving the mapping induced by the subroutine on its arguments and any global variables referenced in the body of the code. Thus arcs in other routines representing calls of this routine may have this mapping appended to their annotations. The derived input/output relation for the subroutine already analyzed may now be used in analysis of calling subroutines. The only case needing special attention is that of mutual or self recursion, where a state variable representing a stack frame must be introduced, and the recursive definition left incompletely evaluated so as to avoid an "f(f(......(x)...))" kind of result.

Within this model an application program is simply a subroutine called by the operating system, and its input/output relation can be found just like any other subroutine. The operating system is special in that it has a non-terminating model showing periodic motion that leaves the machine state invariant, and temporarily exits from this to execute application programs.

7.3. The Complexity of an Analysis

Analogous studies of hardware (see Barrow [1983]) "hide" details of subsystems, deriving their input/output relations and using these as surrogates for the systems themselves in analysis at the next level up.

Similarly, if a program can be partitioned into disjoint subgraphs (i.e., subroutines), the combinatorial explosion of path walking on a large graph does not take place. If all subroutines are roughly the same size, the graph walk has roughly the same complexity for each. Even though the symbolic algebra of manipulating input/output expressions becomes more difficult as one moves up in the hierarchy, one more layer of "internal" variables (specified in the "connected" predicates defining the system architecture in terms of its parts) disappears. Hence, the intermediate expression swell is constrained to that absolutely necessary for conformity to specification. Moreover, since these expressions are a formal statement of what the programmer knew all along, their complexity is probably bounded by human ability to write software in the first place.
8. Composite Hardware-Software Systems

The model includes studies of composite systems quite easily. The simplest example is that of a software system driving a pair of input/output ports to communicate with external hardware. Assuming the ports are memory mapped after the style of the Motorola 68XXX processor series, we have

```
module(program).
  port(program, devout(P), output, bit32).
  port(program, dev_in(P), input, bit32).
  module(peripheral)
    port(peripheral, p_in(P), input, bit32).
    port(peripheral, pout(P), output, bit32).
  module(system).
    part(system, progA(Sys), program).
    part(system, perA(Sys), peripheral).
    connected(system, devout(perA(Sys)), pjin(progA(Sys))).
    connected(system, pout(progA(Sys)), dev_in(perA(Sys))).
```

In other parts of the program definition, the appearance of devout or dev_in in variable assignments means that these assignments send or acquire data values to and from the peripheral. But in the peripheral description these are treated as 32-bit-wide arrays of binary signals.

This model is sufficient to allow analyses where the processor description is abstracted so that it deals only with those dataflows relevant to the peripheral design, while the peripheral hardware is described in detail. These kinds of analyses assume that the properties of the CPU and compiler are as described by the vendor (principally on the grounds that a large user community would have reported any failures) and concentrate on proofs about a complete system unique to some particular small user community. This is exactly the balance needed to assure system quality: the unique features not being tested by others in day-to-day use of a processor receive detailed attention, while those features where a large peer group is in agreement about conformity to specification receive less attention.

9. Applications to System Performance Studies

The program flow graph, with annotations describing changes induced in program state by passage along each arc, is a completely general model of program execution on both von Neumann and other architectures. In fact, because it is founded in a discrete model of evolution in time, it suffices to describe any engineered artifact built from digital logic components, even for multivalued logics. Thus it appears to be a valid model for any digital system evolving in time through a sequence of limit states of parameterized ordinary differential equations.

Although most of this report is concerned with input/output relations, the methods can be used to compute any function of path on the flow graph. Thus the methods are a theoretical and algorithmic foundation for study of system performance and the way this depends on
subsystems and their interconnections. The level of detail used is very much a matter of choice and costs. The model \( G'(\text{Lambda, Sigma, M}) \) includes all programs in written in Lambda. If \( G' \) is constrained by a problem domain \( G(L,S,A) \), then \( G' \) includes only problems in that domain. If, in addition, a frequency distribution for sets of input data bound to to variables in \( S \) is given, then the model deals with an ensemble of computations from \( G(L,S,A) \).

Just as the input/output relation for an arc may be determined by the BGC algorithm from the semantics of \( M \), so may the time needed to traverse that arc be determined from the elementary execution times of opcodes in \( M \). If \( M \) is pipelined, so that a unique execution time for an opcode does not exist, the description of \( M \) must be refined to make the significant hardware arbitrators of contention for serially reusable resources part of the model.

Once this has been done, there is—in principle—no problem barring quite precise models of system performance. In fact, of course, we have the twin bugbears of data about jobstreams and intermediate expression swell in the computation. However, verifications of complete CPU designs defined at the MOSFET level are now being done. It seems quite likely that the corresponding analyses of properties of many subroutine libraries are no more difficult. Further, the hierarchical nature of the analysis technique allows us to claim that most applications using these libraries pose no more difficult problems to a symbolic manipulation tool than the libraries did themselves.

10. Problems Still to Be Solved

A colleague remarked of an earlier draft that it did not mention the truly difficult problems. They are all problems of scale. Once it is established that the space of all possible inputs to a program \( P \) contains only a finite number of points, then there is obviously a finite decision procedure determining the correctness of \( P \), namely, to run it against all data points and examine the output for each. This is impossible in practice, although it is a model of the procedure called software validation.

At the other end of the spectrum from software validation is software verification, which comprises proofs of theorems about programs. One model of this is the algorithms given in this report, which show how to derive an input/output mapping in terms of only the "externally visible" variables of a program from the program that may have many "hidden" variables (in the sense of data hiding in subroutines). Having obtained the input/output mapping, one still must prove it equivalent to the specification. Such proofs of equivalence may be quite difficult, depending on whether a canonical form exists for the mapping. And even if a canonical form does exist, reduction to the canonical form may be a lengthy task. A number of points are worth making about this question.

First there is a well-known difficult example. Several canonical forms exist for expressions in combinational logic [Mange 1986]. De Morgan's theorem allows many other forms equivalent to the canonical one. Reduction of a given form to a canonical form has been found lengthy and difficult in many cases [Smith 1986]. Thus the question arises as to why Barrow was so successful, and why Hunt and Moore [Hunt and Moore 1986] were able to verify a 16-bit CPU, when a 4-bit ALU such as the 74LS181 can cause severe difficulty to a good theorem-proving system. The corresponding question in the case of the model presented here is that there may be more than one equivalent input/output relation for a program, and we have certainly not addressed the question of getting from one to the other.

Clearly if the specification is a canonical form, and there is a set of confluent rewriting rules, then the rules will derive the specification from any input/output relation. The question of the complexity of the rewriting procedure is still open, however, and it could possibly be just as serious a problem as a case analysis on the set of all inputs. We have started work on this question and hope to report the results in a future publication.
Acknowledgments

Our thanks are due to Brian Smith for many discussions of program correctness, to Ross Overbeek some years ago for a discussion of computable numbers, and to Gail Pieper whose work as an editor has made this report more useful than it was when she first received it. Blame for faults in understanding, correctness, and other things, however, rests squarely on the shoulders of the senior author.

References


Distribution for ANL-87-4

Internal:

J. M. Beumer (2)
J. R. Gabriel (10)
H. G. Kaper
A. B. Krisciunas
G. W. Pieper (152)
E. P. Steinberg

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (5)

External:

DOE-TIC, for distribution per UC-32 (133)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
  J. L. Bona, Pennsylvania State U.
  T. L. Brown, U. of Illinois, Urbana
  P. Concus, LBL
  S. Gerhart, MCC, Austin, TX
  J. A. Nohel, U. of Wisconsin, Madison
  D. Austin, ER-DOE
  H. Barrow, Fairchild
  R. Chapman, Wadham College, England (10)
  S. Daniel, Motorola, G.E.G. Phoenix
  J. Kljaich, Bell Labs, Naperville, IL (10)
  G. Michael, LLL
  J. Mills, ASU