Practical Program Verification

Automatic Program Proving for Real-Time Embedded Software

John Nagle
Scott Johnson

Ford Aerospace and Communications Corporation

Abstract

Despite the attractiveness of the concept, attempts to date to use proof of correctness techniques on production software have been generally unsuccessful. The obstacles encountered are not fundamental. We have implemented a proof of correctness system to be used for improving the reliability of certain small, real-time programs. It appears that many of the problems of past systems can be avoided.

This work is supported by the Long Range Research Program of the Ford Motor Company, Dearborn, Michigan.

Philosophy

The first requirement of any reliable program is that it not cease functioning. We are of the opinion that any attempt to perform useful proofs of correctness must begin by insuring that the program does not violate the rules established for valid programs by both the language and the machine. Only when this absence of run-time errors has been established is it meaningful to attempt proof of higher-level constraints.

Were there some way known of specifying the desired behavior of a program in a readable, complete, and concise way, we would have considered verification of the program against some form of formal specification of the program. Devising such a notation is a task of extreme difficulty, comparable to that of designing a very high level programming language. We felt that verification was hard enough without trying to tackle the formal specification problem at the same time. Our system thus accepts only constraints, in the form of procedure entry and exit conditions and module invariants, as additional requirements laid upon the program. These constraints, along with a large number of

constraints required to determine that the program does not perform illegal operations at run time, are the goals of our proofs.

We consider the soundness of the verification to be the responsibility of the verification system, not the users thereof. Where some rule must be enforced to insure the soundness of the verification, the verifier must be capable of mechanically checking for violations of this rule. Over half of our implementation exists to check for violations of rules which would render the verification invalid.

The programs to be verified are written in Pascal-F, a dialect of Pascal developed at the Ford Motor Company's Scientific Research Laboratories. The language is designed to be used in the building of reliable real-time microprocessor control programs. These programs run stand-alone in dedicated microprocessors, and deal directly with hardware interfaces. Pascal-F provides hardware interface and multiprogramming primitives similar to those of Modula I. The Pascal-F Verifier processes the same language processed by the Pascal-F compiler.

It is not permissible to severely restrict the use of the language. Because the programs being verified must execute high speed control loops, our restrictions must not force inefficient coding practices. We have defined and implemented a set of restrictions which appear not to cause the generation of inefficient code, provided that certain specific optimizations are provided in the compiler.

Approach

Our theoretical basis is traditional. [FLOYD67] We use entry and exit assertions, loop invariants, and module invariants. We do not use explicit quantification; functions implying implicit quantification may be introduced and rewrite rules about them defined. [STANFORD79] Our system does not allow the user to introduce new axioms. Such introduction is known to be unsafe [BOYER81]. New rewrite rules are accepted, but only after a proof of the rule has been generated by using a powerful, semi-automatic theorem prover [BOYER79] separate from the verification system proper.

Our goals are similar to those of German [GERMAN81] in that we desire primarily to prove shallow properties of sizable programs, and certain of our techniques are drawn from his work. Our system, although capable of full proofs of correctness, is not normally utilized for that purpose. The primary use of the verifier is to obtain assurance that the program being verified will not engage in gross misbehavior such as subscripting out of range, overflowing, or infinitely looping. This goal governed strongly many of our design decisions.

The proofs performed during the verification process are performed by a decision procedure which is complete within its limited scope and fully automatic. This automatic simplifier handles linear arithmetic, propositions, arrays, records, and type information. [OPPEN79]. The application of rewrite rules is also automatic, but the rules are applied only in restricted circumstances, so that it is generally necessary to construct a specific rule when a rule needs to be applied.

The language system

Pascal-F is a dialect of Pascal developed by E. Nelson at the Ford Scientific Research Laboratories in Dearborn, Michigan. It combines features of Pascal, Ada, and Modula, to provide a language for programming dedicated computers running without an operating system. Code generators exist for the Digital Equipment Corporation LSI-11 and the Ford Electronic Engine Control IV microprocessor (now offered by Intel as the Intel 8061). The basic Pascal-F language has been augmented with statements and declarations needed for verification purposes. The syntax of these statements is integral to the language, i.e. the verification information is not embedded in comments but is expressed in syntax as close as possible to that of normal Pascal. New keywords, such as **ASSERT** and **INVARIANT** have been introduced to make this possible. The compiler and verifier have a common first pass. This first pass processes these statements, syntax and type checking them, and either passes them on to the later passes (if being used as part of the verifier) or discards the information obtained (if being used as part of the compiler).

User interface

We attempt to produce diagnostic messages in the style of a compiler. All error messages reference a source line with which the problem is associated. A typical example appears below.

Our usual demonstration of the system is to present a program which has been verified, and

invite visitors to introduce an error which would cause the program to fail, either by breaking a rule of the language or by violating the assertions in the program text. We then present the modified program to the verifier. We have not yet had it fail to detect an error, and in most cases, the diagnostic messages produced directly reflect the error inserted.

An example

This program fragment is typical of the sort of processing which takes place in small microprocessor control programs. The code here would be suitable for controlling a stepping motor attached to a load of moderate mass, where the motor must be slowed down as the desired position is reached to avoid overshoot and hunting.

stepdrive.pf:

```
 1 {
 2       Verifier Demonstration Program
 3
 4       Stepping motor drive example.
 5
 6                              Version 1.5 of 2/24/82
 7 }
 8 program stepdrive;
 9 const maxint = 32767;                { biggest integer }
10 type
11    dir = (down,up);                 { direction of motion }
12    pos = -5000..5000;               { range of positions }
13    rate = 0..2000;                  { steps/compute cycle }
14    motion = -2000..2000;            { signed rate }
15 {
16    log2  --  extract log to base 2 of number, except at 0 and 1
17
18    Takes in numbers from 0 to maxint/2 and returns numbers from
19    0 to 15;
20
21    Note that the exit conditions of log2 do not fully describe
22    what it does; they just impose some constraints on the result.
23 }
24 procedure log2(var n: integer);
25 exit n <= 15; n >=0;                  { bounds on n }
26     n <= n.old;                    { log2(n) <= n }
27     (n.old = 0) implies (n = 0);       { log2(0) = 0 }
28     (n.old = 1) implies (n = 1);       { log2(1) = 1 }
29     (n.old > 0) implies (n > 0);       { nonzero if nonzero input }
30 entry n >= 0;                        { only on positive numbers }
31      n <= maxint div 2;              { upper bound for n }
32 type smallint = 0 .. 15;
33 var i: smallint;                     { loop counter }
34    log: smallint;                   { resulting log }
35    twotoi: 1..maxint;               { number to double }
36 begin
37    if n <= 1 then log := smallint(n)   { log(0) = 0 by convention }
38                                  { log(1) = 1 by convention }
39    else begin
40       twotoi := 2;                  { 2**i }
41       log := 0;                     { log lags behind i }
42       for i := 1 to 15 do begin       { for maximum needed cycles }
43
44          if twotoi <= n then begin   { if not big enough yet }
45             log := log + 1;         { increment log }
46             twotoi := twotoi * 2;   { double value }
47             end;
48
49          state (defined(twotoi), defined(log),
50                0 < log,
51                log < twotoi, log <= n, log <= i);
52          end;
53       end;
```

```
54    n := log;                    { return log }
55 end {log2};
56 {
57    calcsteprate  --  calculate stepping rate for stepping motor
58
59    Called once per 100ms to calculate stepping rate to be used
60    during next 100ms.
61 }
62 procedure calcsteprate(currentpos,     { current shaft position }
63                desiredpos: pos;     { desired shaft position }
64                var steprate: motion;{ step rate to use }
65                var stepdir: dir);   { direction to step }
66 exit
67    (currentpos.old = desiredpos.old) implies
68       (steprate = 0);               { must stop }
69    (currentpos.old <> desiredpos.old) implies
70       (steprate <> 0);              { no stall }
71    (currentpos.old > desiredpos.old) implies
72       (stepdir = down);             { direction check }
73    (currentpos.old < desiredpos.old) implies
74       (stepdir = up);               { direction check }
75    (stepdir = up) implies           { no overshoot }
76       (currentpos.old + steprate <= desiredpos.old);
77    (stepdir = down) implies         { position }
78       (currentpos.old + steprate >= desiredpos.old);
79 var move: integer;
80 begin
81    move := currentpos - desiredpos;    { steps to goal }
82    stepdir := up;                   { assume upward move }
83    if move < 0 then begin           { if downward direction }
84       stepdir := down;              { so note }
85       move := - move;               { make move positive }
86       end;
87    log2(move);                      { reduce exponentially }
88    steprate := motion(move);        { rate from log2 }
89    if stepdir = down then           { if down direction }
90       steprate := -steprate;        { step other way }
91 end {calcsteprate} ;
92
93 begin {main}
94 {No main program, just a demonstration of procedures this time.}
95 end.
```

The example contains a bug. The verifier's diagnostic messages appear below.

```
Verifying calcsteprate
Could not prove {stepdrive.pf:73}
   (currentpos.old < desiredpos.old) implies (stepdir = 1)
(exit assertion)
for path:
{stepdrive.pf:67} Start of "calcsteprate"
{stepdrive.pf:83} IF->THEN

Could not prove {stepdrive.pf:71}
   (currentpos.old > desiredpos.old) implies (stepdir = 0)
(exit assertion)
for path:
{stepdrive.pf:67} Start of "calcsteprate"
{stepdrive.pf:83} IF not

2 errors detected

Verifying log2
No errors detected

Verifying stepdrive
No errors detected
```

Two errors were diagnosed. Both errors represent failures to verify an exit condition of the procedure *calcsteprate.* Each message references the source file name *(stepdrive.pf)* and source line number for the proof goal, and contains a copy of the proof goal itself. The path being traced is displayed by stating the branch points encountered along the path and giving the branch taken at each conditional. Note that the values *down* and *up* are now represented as 0 and 1. Since enumerated types are represented in the real machine as integers, we do our verification work on them as integers.

Our experience is that the information shown here is enough to make it possible for the programmer to find the problem, without recourse to examining the verification conditions themselves. The reader is invited to find the bug in the example.*

Note that nothing was printed about the many proofs which succeeded. Only the items requiring attention by the user appear.

There were 49 goals identified by the system which need to be proved. Each of these is a potential trouble spot in the program.

Eighteen proof goals were generated to insure that variables are defined at the moment of reference. This is a check for reference to uninitialized variables. In the cases below, all the variables are simple ones (not array or record variables) but the verifier would insist, at an array reference, that the specific element being referenced be previously initialized and provably so. The number in braces is the relevant source program line number.

```
{37} "n" is defined
{37} "n" is defined
{44} "twotoi" is defined
{44} "n" is defined
{45} "log" is defined
{46} "twotoi" is defined
{49} defined(twotoi)
  (STATE assertion)
{49} defined(log)
  (STATE assertion)
{54} "log" is defined
{24} defined(n)
  (exit assertion)
{83} "move" is defined
{85} "move" is defined
{87} defined(n)
  (entry assertion of "log2" {24} )
{88} "move" is defined
{89} "stepdir" is defined
{90} "steprate" is defined
{62} defined(steprate)   (exit assertion)
{62} defined(stepdir)   (exit assertion)
```

Twelve goals were generated by operations involving subrange types of Pascal. Note that the type **integer** is defined as the subrange

---

* Hint: examine line 81.

-32768..32767 in this implementation. No checks were generated for overflow of intermediate operations in this example, because the ranges of variables happened to be such that either intermediate result overflow was not possible or the check was subsumed by the subrange check associated with the left side variable of an assignment statement. Had such checks been necessary, they would have been generated.

```
{37} n >= 0
  (range check for "log" 0..15)
{37} n <= 15
  (range check for "log" 0..15)
{45} log + 1 >= 0
  (range check for "log" 0..15)
{45} log + 1 <= 15
  (range check for "log" 0..15)
{46} twotoi * 2 >= 1
  (range check for "twotoi" 1..32767)
{46} twotoi * 2 <= 32767
  (range check for "twotoi" 1..32767)
{81} desiredpos - currentpos >= -32768
  (range check for "move" -32768..32767)
{81} desiredpos - currentpos <= 32767
  (range check for "move" -32768..32767)
{88} move >= -2000
  (range check for "steprate" -2000..2000)
{88} move <= 2000
  (range check for "steprate" -2000..2000)
{90} - steprate >= -2000
  (range check for "steprate" -2000..2000)
{90} - steprate <= 2000
  (range check for "steprate" -2000..2000)
```

Four goals were generated by the loop invariant (the **STATE** statement.)

```
{49} 0 < log    (STATE assertion)
{49} log < twotoi    (STATE assertion)
{49} log <= n    (STATE assertion)
{49} log <= i    (STATE assertion)
```

One check was generated to insure lack of overflow of a FOR loop control variable. (In this case the loop bounds were constant, so the check is redundant, but had they been they variable a check would have been required).

```
{42} i <= 14    (FOR loop count)
```

Finally, there are fourteen entry and exit assertions. These have the meaning usual in verification; the entry assertions are checked at the start of a call and assumed true at the beginning of the procedure, and the exit assertions are checked at exit from the procedure and assumed at return from the call.

```
{87} n >= 0
  (entry assertion of "log2" {30} )
{87} n <= 16383
    (entry assertion of "log2" {31} )
{25} n <= 15
    (exit assertion)
{25} n >= 0
    (exit assertion)
{26} n <= n.old
    (exit assertion)
{27} (n.old = 0) implies (n = 0)
    (exit assertion)
{28} (n.old = 1) implies (n = 1)
    (exit assertion)
{29} (n.old > 0) implies (n > 0)
    (exit assertion)
{67} (currentpos.old = desiredpos.old)
          implies (steprate = 0)
    (exit assertion)
{69} (currentpos.old <> desiredpos.old)
          implies (steprate <> 0)
    (exit assertion)
{71} (currentpos.old > desiredpos.old)
          implies (stepdir = 0)
    (exit assertion)
{73} (currentpos.old < desiredpos.old)
          implies (stepdir = 1)
    (exit assertion)
{75} (stepdir = 1) implies
      (currentpos.old + steprate
      <= desiredpos.old)
  (exit assertion)
{77} (stepdir = 0) implies
      (currentpos.old + steprate
      >= desiredpos.old)
  (exit assertion)
```

Even in a modest program, the number of possible trouble spots is quite large. Any one of these spots has the potential of causing a major failure of a program. We look upon this kind of analysis as the software engineering counterpart of structural stress analysis for buildings and other engineered structures.

Economics

Our system runs on a VAX 11/780, a ‘‘super-mini’’. Running time for the above example, without previous results being available, is about seven minutes. This is not unreasonable for the applications envisioned for the programs being verified. Reverifications after minor changes are faster. The example shown requires 141 proofs.

A separate proof is attempted for every proof goal for every path. No user interaction was required during the seven-minute verification.

Implementation

The complexity of a proof of correctness system in which the semantics of the language are not grossly restricted is considerable. The Pascal-F verifier is roughly of the complexity of an compiler with extensive global optimization. The system is divided into two separate subsystems. The verifier, which takes Pascal-F programs and attempts to verify them, is entirely automatic and provides for no user interaction once started. The rule builder, which is used to prove the validity of new rules, is a version of the Boyer-Moore theorem prover.

The verifier proper is composed of the compiler pass, the decompiler, the path tracer, and the simplifier.

The compiler pass takes in Pascal-F and emits I-code and a storage allocation map. The compiler pass used in the verifier is the same as the first pass of the Pascal-F compiler used to generate machine code. This insures that the definition of the language is the same for the verifier and the compiler.

The decompiler takes in the I-code and the storage map, constructs a code tree, annotates the code tree with variable names by using the storage map, constructs variable set-used lists and procedure call graphs, performs transitive closure to determine the side effects of each callable routine, and generates output in the form of an assertion language in which the semantics of the program statements have been expressed as assertions placed on a flow graph. In the assertion language, which we call J-code, all statements, including procedure and function calls, are expressed as a list of variables changed along with the assertions true after the statement is executed.

The path tracer, or verification condition generator, takes in the assertion language text and traces out all possible paths by which a proof goal may be reached in the program. Verification conditions are generated for each possible path to each goal, and submitted to the simplifier, which is called as a subroutine. When a proof attempt fails, the path tracer generates an appropriate diagnostic for the user.

The simplifier is based on D. Oppen’s earlier simplifier. An improved propositional mechanism using tableaus, a type definition facility, and a rule handler have been added.

The compiler, decompiler, and path tracer are written in Pascal; the simplifier is in Lisp. The verifier proper is composed of about 22,000 lines of source code.

The rule builder is a version of the Boyer-Moore theorem prover initialized with a theory compatible with that built into the simplifier. The rule builder is run separately from the remainder of the verifier, and communicates with the simplifier by placing rewrite rules in a file associated with a specific program being verified. There is no communication from the verifier to the rule builder. The rule builder is an Interlisp program.

Capturing the semantics of the language

The language which we verify is essentially taken to be defined by its compiler. The compiler is a straightforward recursive-descent compiler which emits a reverse-Polish language suitable for direct execution on a suitable machine. There is an interpreter for this language, but normally the translated program is passed to a code generator for a target machine.

The reverse-Polish language is intended for execution by a stack machine. Data types are integers of various bit widths, arrays, and binary fixed-point numbers. Pointers are used only for parameter passing and for implementing Pascal WITH statements. Typical operators in this language are

IADD	(Integer addition) Pop two operands from the stack. The operands are 16-bit twos complement integers. Add these operands with twos complement 16-bit signed arithmetic, and push the result on the stack. It is an error if the result is not in the range -32768..32767.
STOL w	(Store). Pop an address from the stack. Pop w bits from the stack. Store the w bits at the

address obtained.

VARBL w a	(Variable address take). Push the address a on the stack. The object being addressed is w bits wide.
INDEX w n	(Array index). Pop an address from the stack. Pop a 16-bit value, which must be nonnegative, from the stack. Multiply the value by the item width w, then add the address popped. Pop w bits from the stack. Store these bits at the address indicated.

The reverse-Polish language, which we call I-code, contains 108 operators. The usual operators for machine arithmetic are provided, along with IF, FOR, LOOP, and CALL operators for control. There are no unstructured branching operators. Each operator has semantics similar in complexity to the ones above.

As an example,

```
A[i] := x + y;
```

with the storage allocation map

x	address 20, width 16 bits
y	address 40, width 16 bits
i	address 60, width 16 bits
A	address 100, width 16*100 bits (array 0..99)

would be translated into

```
VARBL    16 20   (Take value of x, which is 16 bits)
VARBL    16 40   (Take value of y, 16 bits wide)
IADD             (Add x and y)
VARBL    0 100   (Take address of A, no width)
VARBL    16 60   (Take value of i, width 16 bits)
INDEX    16      (Compute address of A[i])
STOL     16      (Store 16-bit result into A[i])
```

We refer to this technique as the *concrete semantics approach*. This very low-level representation of programs has rather well-defined semantics. We choose to work through this low level notation because it gives us a rigorous definition of the exact semantics of machine arithmetic to be performed during the execution of any given program.

Since this notation does not even use variable names, working entirely with addresses and offsets, it would seem that any verification results would be completely incomprehensible to the user. This being unacceptable, we decompile the I-code representation into a new symbolic representation of the program. As we do so, we make note of the constraints imposed by the domain and range of each machine operator. The restrictions of the implementation as to size of operands are thus captured.

J-graphs and J-code

We divide the problem of generating verification conditions into two parts. We first translate the program to be verified into a representation we call a J-graph, and then trace out the paths in the J-graph. We have defined a textual language we call J-code which is used to represent J-graphs. The decompiling pass of the verifier generates J-code from I-code. The path tracer then parses J-code and builds a J-graph, then traverses the J-graph to produce verification conditions.

The J-graph is best thought of as a nondeterministic loop-free program. J-graphs contain the primitives **NEW, SPLIT, WHEN, BRANCH, JOIN, HANG, and REQUIRE.**

The NEW instruction is a nondeterministic, simultaneous assignment statement. The format of a NEW instruction is

NEW <variable-list> <formula>.

When a NEW instruction is executed, the variables in the <*variable-list*> **are updated in such a way as to make the** <*formula*> **true. If** *v* **is a formula in the variable list, the notation** *v.old* **may be used to denote the value of the variable before the** *NEW* **instruction is executed.**

Assignment statements and procedure calls are translated to *NEW* **instructions in a straightforward fashion. For example, the assignment statement:**

X := X + 1

is translated to:

NEW (X) (X = X.OLD+1)

To translate a procedure call, the decompiler determines every variable which could be modified as a result of the call, either by direct modification or through side effects. This list of variable forms the <variable-list> of the NEW instruction. The <formula> is derived from the EXIT condition declared with the procedure.

The SPLIT, WHEN, BRANCH, and JOIN instructions are used to model flow of control in J-graphs. SPLIT and WHEN are used when the flow of control forks off in different directions, and BRANCH and JOIN are used when it comes back together again. This notation merely allows us to represent a loop-free flowchart in a linear fashion. As an example,

IF X > Y THEN M := X ELSE M := M + 1;

would be represented by

```
SPLIT    1
WHEN X > Y    1
NEW (M) (M = X)
BRANCH    2
WHEN NOT (X > Y)    1
NEW (M) (M = M.old + 1)
BRANCH 2
JOIN 2
```

All similarly numbered statements are considered to be connected.

The REQUIRE statement is used to put verification goals into J-graphs. The format of a REQUIRE instruction is

REQUIRE <formula>

Whenever we must prove that some formula is always true when control reaches some point in a program, we put a REQUIRE instruction containing that formula at the corresponding point in the J-graph. For each REQUIRE statement, the path tracer will trace out all paths from the REQUIRE statement back to the beginning of the program, and for each path, it will generate and submit to the theorem prover a verification condition.

Finally, the instruction terminates a path. An execution path which reaches a HANG instruction is not continued past that instruction.

J-graphs must be loop-free. Every execution path which starts at the beginning of the J-graph must be traced forward to a HANG instruction without reaching any instruction in the graph more than once. Since we require users to write loop invariants for all loops, we are able to break the loop at the invariant and thus end up with a loop-free graph. For the program fragment

```
REPEAT
  s1;
  INVARIANT p;
  s2;
  UNTIL b;
```

where *s1* and *s2* are blocks of code, *p* is the loop invariant, and *b* is a boolean expression, the corresponding J-code is

```
SPLIT   1
WHEN TRUE   1
BRANCH   2

WHEN TRUE   1
NEW (<list-of-variables-changed-in-loop>) p
<J-code for s2>
SPLIT   3
WHEN NOT b   3
BRANCH   2

JOIN   2
<J-code for s1>
REQUIRE p
HANG

WHEN b   3
```

(The graph has been simplified for clarity by the omission of the statements usually present for proof of loop termination.) The loop invariant is proved by induction on the number of times the loop body is executed. For the base case of the induction, we must show that *p* is satisfied when the **INVARIANT** statement is reached on the first execution of the loop body. This case is handled by the path which goes back from the REQUIRE through the J-code for s1, the **JOIN 2, the first BRANCH 2 and back to the beginning of the program. For the induction step, we must show that if *p* holds when the INVARIANT statement in the original program, then when the INVARIANT is reached again, P will still hold. This case is handled by the path from the REQUIRE back through the JOIN 2, the second BRANCH 2, WHEN NOT b, the J-code for s2, the NEW statement, and back through the WHEN TRUE 1 and SPLIT 1 to the beginning of the program. Finally, the path taken by the final execution of the loop body is represented by the path starting at the end of the J-code fragment and continuing back via the WHEN b 3 statement, through the loop invariant in the NEW statement, and eventually back to the beginning of the program.**

The J-graph generated thus captures the semantics of the original program. It is worth noting that our implementation generates J-code by a process very similar to that used for generating machine code in a compiler. The process of generating J-code is much more amenable to such treatment than that of directly generating verification conditions, and allows us to draw heavily on techniques from compiler technology in our verifier implementation.

Verification condition generation

The semantics of the language having been captured in the J-graph, the task of the verification condition generator is primarily that of tracing out all paths back from every REQUIRE statement back to the beginning of the J-graph and generating verification conditions during the process. The verification condition formally expresses the proposition ‘‘If this path is taken through the program then the formula on the **REQUIRE** instruction will be satisfied.‘‘. As soon as the path tracer generates a verification condition, it passes the formula to the theorem prover. If the theorem prover cannot simplify the formula to TRUE, the path tracer displays a diagnostic message. The user is not ordinarily exposed to the verification condition itself. Our experience is that diagnostic information of the

form shown in the example above is usually sufficient to allow the user to correct the problem.

The actual verification condition is constructed from formulae on the **WHEN, NEW, and REQUIRE statements and the path. The first step of the construction involves renaming of variables. A unique name is invented for each of the variables in the variable lists of NEW instructions along the path. Then one of these unique names is substituted for every variable in every formula on the path. If** *v* **is a variable in a formula, to find the unique name to be substituted for** *v,search back along the path for the first* **NEW** *instruction that has v in its variable list. The unique name associated with v in that instruction is used.*

Special attention is required when v occurs in both the variable list and formula of a **NEW** *instruction. Recall that v refers to the value of the variable after the* **NEW** *is executed, and v.old refers to the value of the variable before the* **NEW** *is executed. Therefore, the unique name for v is taken from the* **NEW, while the unique name for** *v.old* **is taken from the previous NEW instruction on the path mentioning** *v* **in its variable list.**

After all the variables have been renamed, the verification condition is simply

p1 and p2 and ... pn implies q

where *q* **is the formula in the REQUIRE being processed, and the terms** *p1 pn* **are the formulas on all the other instructions on this path. It is not strictly necessary to include among the** *p* **terms formulas from REQUIRE instructions passed over during backwards tracing, but we do so to stop the user from getting an avalanche of diagnostics that all result from a single error. In effect, after a REQUIRE is passed, it is assumed to be true. This is valid since the graph is loop-free. It is also very effective in reducing the number of diagnostic messages and in speeding up the proofs.**

Optimizing verification conditions

The major motivation for our approach to verification condition generation is the ability to provide good diagnostics. However, the generation of a separate verification condition for every **REQUIRE/path pair is expensive, since the number of paths is exponential in the size of the graph. Fortunately, we have discovered some techniques to reduce the amount of computation required to process a J-graph.**

The first optimization we call *NEW balancingbe NEW balanced if for every variable v and point P in the graph, every path from the beginning of the graph to P contains the same number of* **NEW** *instructions with v in their variable lists. If a graph is NEW-balanced, the path tracer does not have to perform a renaming operation each time it processes a verification condition. Instead, renaming need only be performed only once, before any verification conditions are generated.*

The second optimization is a technique for eliminating irrelevant clauses in verification conditions. Consider the program fragment

```
x := x + 1;
y := y + 1;
assert(y > 0);
```

The J-graph generated for the program fragment above will contain two **NEW** *instructions and a* **REQUIRE** *instruction. The verification condition generated for the* **REQUIRE** *must make use of the* **NEW** *instruction for y, but it could ignore the one for x, because it has no effect on the assertion. An effective technique for performing this optimization has been implemented.*

Theorem proving

Providing a sound theorem-proving system of reasonable speed is quite a challenge. We use two theorem provers, a descendant of Oppen's simplifier and an essentially-unmodified Boyer-Moore structural induction prover. Both have been described extensively elsewhere. [BOYER79], [OPPEN79]. The Oppen prover is an integral part of the verifier but the Boyer-Moore prover is run separately to produce rules for the Oppen prover. This rather unusual approach, although somewhat difficult to implement, appears to be a productive line of development.

The Boyer-Moore structural induction prover is quite powerful but slow, and tends to become lost if the formula being proven has a large number of irrelevant terms. The Oppen prover is much faster, and a complete decision procedure in its restricted universe, but not powerful enough to allow proofs of anything involving iteration or induction. We thus feed the Oppen prover the verification conditions and let the user talk directly to the Boyer-Moore prover. The Oppen

prover has been augmented with a rather simple rule handler which applies rules exhaustively, but only one deep. The intent here is not to provide a general rule-based proof system but merely to make it possible to apply general theorems proven in the Boyer-Moore system as rules in the Oppen system.

We allow users to define new functions and predicates as Boyer-Moore recursive definitions. For example, for a proof of a sorting program, we might introduce a definition of *ordered* by the following steps. The user defines *ordered* in Pascal-F as an uninterpreted function (which we call a I "rule function"). The user then defines *ordered* to the Boyer-Moore system. (The notation here is that of Boyer and Moore, with the addition that *selecta!* is the array subscripting operator.)

```
Definition.
(ordered A I J)
  =
(IF (LESSP I J)
   (AND (NOT (LESSP (selecta! A (ADD1 I))
                    (selecta! A I)))
        (ordered A (ADD1 I) J))
   T)
```

This recursive definition says that *ordered(A,I,j)* is true if *J* is less than *I*, otherwise *ordered* is true if and only if both *A[I] < A[J] + 1 ordered* is true from *I+1* to *J*.

Once this definition has been introduced and found sound by the Boyer-Moore prover (which insists that such recursive definitions provably terminate), we can prove some theorems about *ordered.*

```
Theorem.  ordered.void.rule:
(IMPLIES (AND (NUMBERP I)
              (NUMBERP J)
              (arrayp! A)
              (LESSP J I))
         (ordered A I J))
```

(i.e. if *J* < *I*, *ordered* is vacuously true),

```
Theorem.  ordered.single.rule:
(IMPLIES (AND (arrayp! A) (NUMBERP I))
         (ordered A I I))
```

(a single element array is always ordered),

```
Theorem.  ordered.unchanged.rule:
(IMPLIES (AND (NUMBERP I)
              (NUMBERP J)
              (NUMBERP X)
              (arrayp! A)
              (OR (LESSP X I) (LESSP J X))
              (ordered A I J))
         (ordered (storea! A X V) I J))
```

(storing into an array outside the limits of the ordering does not destroy the ordering), and

```
Theorem.  ordered.extend.downward.rule:
(IMPLIES (AND (NUMBERP I)
              (NUMBERP J)
              (arrayp! A)
              (ordered A (ADD1 I) J)
              (NOT (LESSP (selecta! A (ADD1 I))
                          (selecta! A I))))
         (ordered A I J))
```

(the ordering may be extended downward one element if the new element is less than the previous first element of the ordering).

All of the above rules are provable by the Boyer-Moore prover as written above. Once the rules needed for proving the program have been proven with the Boyer-Moore prover, the user then invokes a utility program which reads the Boyer-Moore database of proved theorems and definitions, finds all rules, (our convention is that any Boyer-Moore theorem with a name ending in *.rule* is to be transferred to the Oppen system) and translates them into the form needed by the Oppen prover's rule handler. The original verification is then rerun. Where applicable, the rules will be invoked automatically, and the verification conditions will be proved.

The process is not as automatic as it appears. Although there is no user interaction during proofs in either system, both systems often require help. In the Boyer-Moore system, one provides help by suggesting lemmas which help the prover to prove the rules needed for the program proof. This form of manual subgoaling is often difficult. One is required to work strictly upward, devising lemmas which are both provable by the automatic prover and lead the prover toward reaching the desired goal. Simple recursive properties of arrays, such as the *ordered* example above, are within the power of the prover to handle without help, but more complex properties can be difficult.

Once rules deemed sufficient to prove the program correct have been generated, the problem is not over. Since rules are invoked only one deep (a rule will not be applied to a form created by another rule), the user may have to provide some assistance to the rule handler in the main verifier. Again, this is done by adding intermediate proof goals. In the Pascal-F source program, one adds **ASSERT** statements immediately ahead of the statement generating the proof goal with which one is having trouble. It is always the case that

Pascal-F source of the form

```
ASSERT p;
ASSERT q;
```

will generate a verification condition for which the rule handler would apply a rule of the form *p implies q, if such a rule were present. Further, once an assertion has been proven, the verifier will assume its truth in Pascal-F code following the assertion. Thus, when faced with a difficult proof, one adds ASSERT statements to force the verification conditions into forms which the rule-handler will recognize as immediately provable from a known rule.*

We thus have adequate power to handle difficult problems. We usually hold most of this power in reserve, and attempt to prove only simple properties of moderate-sized programs. The level of expertise required to push a difficult proof through the system is presently rather high compared to the formal mathematical knowledge of most working programmers. Our current thinking is that programmers will use the verifier with a database of previously-proved rules, obtaining the aid of a verification expert when a new rule is necessary.

Insuring the soundness of the two-prover approach is non-trivial but possible given the power of the Boyer-Moore system. If the semantics of a function were inconsistent in the Boyer-Moore and Oppen systems, unsoundness would result. It is necessary to prove that the definitions in the Boyer-Moore system are consistent with the axioms built into the Oppen system. Such proofs are necessary for any function symbol known to both systems. The function symbols so known include only the primitives of arithmetic, the Boolean connectives, the array operators select and store (which we call selecta! and storea!), and some type predicates such as arrayp! (true for array-valued variables) and numberp! (true for the natural numbers). Machine proofs of the axioms of the Oppen system have been generated using the Boyer-Moore system initialized with our definitions. This crosscheck insures that our definitions of the Oppen objects in the Boyer-Moore system are consistent with the Oppen built-in rules. Once the built-in theories have been shown to be consistent, we can allow the user to introduce new functions in the Boyer-Moore system, but such functions will be uninterpreted functions in the Oppen system and thus cannot introduce contradictions.*

Directions for further work

Our immediate plans call for using the verifier on a pilot project to determine the effectiveness of these tools on certain types of embedded systems. We may then consider further tool development.

We have a number of ideas for obtaining substantial speed increases in the automatic proof process. We will hint at one of these. By integrating path tracing with theorem proving, we can avoid the generation of verification conditions in the traditional sense and take advantage of the fact that, on many paths, we need not trace back all the way to the beginning of the program before discovering that the proof goal has been satisfied.

Central to this scheme is the fact that the Oppen combination of methods stores the state of a proof in a form which can be pushed and popped. One could, in theory, start by providing the prover with the proof goal, and then tracing backwards through the J-graph, add information to the proof system as each assertion is passed. If a proof succeeds before the head of the program has been reached, one can stop tracing the path at that point. Not only has one verified the proof goal for the path being traced; one has verified the proof goal for all paths which include the part of the path being traced which was actually explored.

Even better, one need not throw away all the information now inside the prover to trace the next path. One need only back down the J-graph to the last **JOIN** point, popping out assertions from the prover state as they are passed over while backing down the tree. When the back-down process reaches the original proof goal node, the proof is complete. In many cases, this will occur long before the entire sheaf of paths leading to the proof goal has been examined.

This line of thinking leads us to suspect that future program verification systems will not generate verification conditions at all, but will operate on graph representations of the program similar to our J-graph. Other benefits besides speed may be obtainable in this way, including improved diagnostic messages and more source-program oriented statements about why the proof is failing.

---

* by Dr. John Privitera of Ford Aerospace

Conclusions

1. Proof of correctness is currently usable as a technique for improving the reliability of small real-time programs.
2. The machine resources required for machine proof, while substantial, are not unreasonable.
3. Automatic proof systems must, to be sound, refuse to accept new user provided lemmas without proof.
4. A verification system should operate on the assumption that most of the programs fed to it will initially be incorrect.
5. Adequate user diagnostics are not difficult to implement, but must be implemented if the system is to be usable.
6. When each requirement to be proven generates a separate verification condition for each path, and the diagnostic messages produced for a proof failure include the requirement which could not be proven and the path for which it could not be proven, users are able to find the problem without examining the verification condition.
7. It is sometimes easier to capture the semantics of the language from a compiler intermediate form rather than trying to axiomatize the source language.
8. Usable verification systems are not small systems; we conclude that constructing a verifier for a given language is a task of roughly equal difficulty with constructing a good optimizing compiler for that language.

References

BOYER79 Boyer, Robert S, and Moore, J. Strother, *A Computational Logic, Academic Press, New York, 1979.*

BOYER80 Boyer and Moore, private communication.

FLOYD67 Floyd, Robert., Assigning Meanings to Programs, Mathematical Aspects of Computer Science, Proc. Symp. Applied Math. Vol XIX American Mathematical Society, Providence, R.I. 1967

GERMAN81 German, S. M., Verifying the Absence of Common Runtime Errors In Computer Programs, PhD Thesis, Harvard University, 1981.

OPPEN79 Oppen, Derek, Simplification by Co-operating Decision Procedures, Computer Science Department, Stanford University, 1979.

STANFORD79 Luckham, German, v. Henke, Karp, Milne, Oppen, Polak, Scherlis, Stanford Pascal Verifier User Manual, Computer Science Department, Stanford University, 1979.