Pascal-F Verifier User's Manual

Version 2

by Scott D. Johnson and John Nagle

Pascal-F Verifier User's Manual

Version 2

by Scott D. Johnson and John Nagle

Ford Aerospace &
Communications Corporation
Western Development
Laboratories Division

3939 Fabian Way
Palo Alto, California 94303

This work was supported by the Long Range Research Program of the Ford Motor Company, and was carried out at Ford Scientific Research Labs in Dearborn, Michigan and Ford Aerospace and Communications Corporation's Western Development Laboratories in Palo Alto, California.

Printing of 11/6/105.

This is the user's manual for the second release of the Pascal-F Verifier. The current version of the system operates on VAX and SUN systems running Berkeley UNIX.

Comments and trouble reports should be addressed to

Division Software Technology and Support
Mail Station X20
Ford Aerospace and Communications Corporation
3939 Fabian Way
Palo Alto, CA 94303.

or

verifier@FORD-WDL1.ARPA

on the Internet.

1. An introduction to verification

Verifiers are not yet common software tools. For this reason, a substantial amount of explanation is in order. This manual is addressed to the professional programmer involved in the production of high-reliability software. Acquaintance with Pascal-F is assumed; acquaintance with verification is not. No specific mathematical background beyond that necessary to comprehend a Pascal-like language is assumed. Users who are uncomfortable with formal mathematics in any form, however, will find using the Verifier rather heavy going.

There has been a certain mystique associated with verification. Verification is often viewed as either an academic curiosity or as a subject incomprehensible by mere programmers. It is neither. Verification is not easy, but then, neither is writing reliable computer programs. More than anything else, verifying a program requires that one have a very clear understanding of the program's desired behavior. It is not verification that is hard to understand; verification is fundamentally simple. It is *really* understanding programs that can be hard.

1.1 What a verifier is and what it can and cannot do

A verifier is a computer program, or set of computer programs. It examines other programs and tries to prove that they meet specified criteria. Some of these criteria are implied by the language in which the program is written. Others are supplied by the programmer or the system designer. It is the task of the verifier to try to show that the

program always meets the stated criteria, no matter what data or conditions the program is faced with, provided only that the program is faithfully executed by the computer.

This task is not an easy one. It is much more difficult than, for example, checking that a program is syntactically correct. Showing that the program works for all cases requires a much more powerful approach than testing, simply because for programs of any complexity, exhaustive testing requires numbers of test cases that are far too large for there to be any hope of trying all of them.

Verifiers attack this problem by turning a program and its criteria for correctness into a large number of mathematical formulas, and then trying to prove that all these formulas always hold. When the criteria for correctness can be expressed mathematically, this approach is very useful. When the criteria for correctness are not easy to define, the approach is of limited use.

Fortunately, it is quite straightforward to express mathematically the concept of a program ‘‘blowing up at run-time.’’ Errors that a run-time system for a language can catch, such as subscripting out of range, exceeding the allowed range of a variable, and referencing a variable that has not yet been assigned a value, are easy to define mathematically. Unlike a run-time system, which can only detect these errors when they occur, a verifier can detect these errors *before* they occur.

The first step in any sound verification is to show that no run-time errors occur. Some verifiers do not bother with this step, but assume, for example, that integer variables can contain any value and that arrays have infinite dimensions. While such verifiers are useful as research tools, one cannot place much confidence in a result of ‘no errors found’ from such a verifier.

Ideally, verifying that a program cannot generate a run-time error should not require any help from the programmer, since the program contains sufficient information to decide this question. However, our verification tools are not sophisticated enough to do this checking without help.

The first step of verification is the translation of the question of the correctness of a program into a collection of mathematical formulas. This process is complex, but it is no worse than that of translating the program into object code, which is performed by a compiler. The hard part of verification is constructing proofs that the formulas are true, or discovering that they are false.

Early attempts at verification required users to write out proofs by hand, just as mathematicians have done for centuries. Other people read over the proofs and checked them. This approach didn’t work very well. It worked about as well as desk checking of programs does. A number of supposed program proofs published in various professional journals have been found to contain errors.

The next step was automatic proof checking. In this mode, people worked out their proofs at computer terminals, with the computer checking each step of the process. This approach is reliable, but it is slow and expensive. It has not been much used because of the high labor cost.

Fully automatic generation of proofs is the ultimate goal. Success has been achieved for

certain kinds of problems. Researchers are trying to extend the range of problems that can be handled automatically. There are good, fast techniques for a useful class of simple problems. Other more powerful (though much slower) techniques can be used on a larger class of problems. Research continues in this area.

A number of automatic theorem proving programs have been written. All verification systems in serious use today use automatic theorem provers. However, verifiers are usually unable to prove programs correct without substantial help from the programmer, because of limitations of these theorem provers. It is this fact that keeps verifiers from coming into widespread use.

Fortunately, the mathematics in real-time programs and system programs tends not to be very advanced. More statements look like

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

than

```
x := (-b + sqrt(b**2 - 4*a*c)) / (2*a);
```

This fact makes verification of these programs tractable even with today's theorem provers.

We can attempt to verify any condition that can be expressed in the form of a computable Boolean expression that is always supposed to be true at some point in the program. Such conditions are called *assertions.* A typical assertion would be

```
ASSERT x > y;
```

One can compare this with a run-time check of the form

```
IF NOT (x > y) THEN ABORT;
```

If correct operation of the program requires that x be greater than y when control reaches this statement, we would like to be convinced that the ABORT will never occur. A verifier can often generate a proof that the assertion is true whenever control reaches the ASSERT statement.

Verifiers generally are not very good at diagnosing why a program cannot be verified. When a verifier says a program is correct according to its rules, then either the program is correct or the verifier is not working properly. (Sadly, an error in the verifier, as well as a compiler error resulting in incorrectly generated object code, are always possibilities.) However, when a verifier fails to prove a program correct, the program may not be in error. It may simply be that the verifier needs more help from the programmer, or that some of the help already given is misleading. It is not always possible for the verifier to diagnose the problem in a concise form. But the verifier can usually point out which section of code is giving trouble. Beyond that, the typical verifier merely lets the programmer see the formula that it is trying to prove and lets the programmer try to figure out what is wrong. This level of diagnostic can be annoying, but is really no harder than

debugging.

Verification is an iterative process. One submits the program to a verification system, gets some error messages, fixes the errors, and tries again until all the errors are gone. Because verifiers tend to be rather slow, facilities for reverifying only the parts that have changed are often provided. This feature is akin to being able to recompile only part of a program after making a change, and speeds up debugging substantially.

In time, verification may become a routine part of programming. At present, it is an area of active research, but the techniques of verification have been used only on a few projects. In most cases, the software tools needed to use verification on real projects have been lacking. This fact has retarded the acceptance of verification as a means of improving the quality of programs.

1.2 The Pascal-F Verifier and its powers

The Pascal-F Verifier is designed to be used to improve the reliability of medium-sized real-time programs written in Pascal-F. Its power is generally adequate to check programs for absence of run-time errors. Higher-level constraints may be also be submitted for verification, and an attempt will be made to verify them, but there are limits on the complexity of the relations that can be verified.

The Verifier operates on a dialect of Pascal-F that has been augmented with language features used to provide additional information to the Verifier. These extensions are described in Chapter 2. The assertions required to verify a program are placed in the program text itself; there is no separate specification file. An extended version of the Pascal-F compiler is available which will accept but ignore the verification statements, allowing verified programs to be compiled without change.

The basic units of verification are the procedure, function, monitor, module, and main program. Collectively, we call these *program units.* We also refer to procedures and functions as *routines.* Each program unit is verified in isolation, using previously stored information about all other relevant procedures and variables. The information stored for each program unit is

- The name, formal argument list, and result type
- The ENTRY and EXIT assertions
- The INVARIANT assertions
- The list of global variables referenced
- The list of global variables altered
- The list of all routines called, and the arguments to each call
- Information concerning multiprogramming.

These items define the *interface* of the routine. The only information about a routine available when verifying its callers is the interface. Therefore all the information about what a routine does must be included in the EXIT assertions, and all the information about what a routine needs must be included in the ENTRY assertions.

Before verifying a program, the verifier checks the program for violations of the rules given in chapter 2. This is referred to as *preverification checking.* Once all parts of a program have passed these tests, which in themselves may show up program ‘‘bugs’’, the actual verification process begins. The Verifier generates assertions for all statements that could cause overflows, out-of-range subscripts, references to variables not yet assigned a value, or other run-time errors. Formulas called *verification conditions* are generated for all these assertions and for the user’s assertions. Attempts are made to prove all the verification conditions using an automatic theorem prover. Diagnostic messages are generated for all unproven conditions.

It is expected that the user will want to alter the program being verified and to attempt reverification. The Verifier maintains a file that minimizes the amount of work required when reverifying a program. The units of reverification are the procedure, function, monitor, and module. In general a change made within a program unit will not require reverification of parts of the program outside that unit unless the interface of the unit is altered by the user.

2. Writing verifiable Pascal-F

Programs to be verified must in a sense be ‘‘understood’’ by the Verifier, for which it needs a substantial amount of help from the programmer. Most of this help is supplied in the form of special statements embedded in the text of the program. These statements are meaningless to the Pascal-F compiler (though the compiler will recognize and ignore them), but to the Verifier they supply information about how the program works.

2.1 The Verifier’s view of Pascal-F

The Verifier needs additional information beyond the Pascal-F statements needed to produce a running program. The basic form of additional information is the assertion. An assertion is a Boolean expression (that is, one whose value is true or false) that is supposed to be true whenever some point in the program is reached. A typical assertion is

```
ASSERT x > y;
```

This assertion is a claim that whenever control reaches the statement, x will be greater than y. The Verifier will take on the task of proving that the code that leads up to the ASSERT statement always guarantees that x is greater than y. Following the ASSERT statement, the Verifier will assume that x is greater than y.

The Verifier does not execute programs. It examines programs and attempts to predict their actions when executed. This examination takes place based on a set of built-in notions about the execution environment.

2.1.1 Initialization

All variables are considered to be uninitialized at start and may not be referenced before being given a value. At program start, only constants have values. Upon entry to a routine, all local variables are uninitialized. Everywhere that a variable is used, the Verifier will attempt to prove that the variable has been assigned a value. For simple

variables this task is usually trivial; for arrays it is usually harder. See the section on the DEFINED predicate for information on how to cope with arrays.

2.1.2 Ranges

All types have finite bounds. For subrange types, these are explicit. The bounds of the type INTEGER are from -32768 to 32767; these bounds are determined by the 16-bit hardware arithmetic used in the implementation of Pascal-F. Everything that has a numeric value thus has a subrange associated with it. All variables that have been initialized are assumed to be within range. Hence, all actions that change the value of a variable must be checked for out-of-range conditions.

2.1.3 Multiprogramming

The Verifier views programs as if control can be taken away from the current process only at certain points called ‘‘singular points.’’ The singular points are at SEND and WAIT statements and places where a process calls a routine exported from a different monitor. This assumption allows the Verifier to process all other parts of programs as if they are purely sequential.

This assumption is a simplification of reality, since processes can be pre-empted when interrupts occur. However, since no monitor is allowed to access a variable that is declared in a different monitor, the fact that there is preemption can be ignored because the preempted process cannot detect that it was preempted.

Normally, the Verifier can assume that any variable not altered by a statement does not change. When a process P encounters a singular point, this assumption is not valid. At those points P can be interrupted by other processes that can alter variables that P is allowed to read. When a process reaches a singular point, monitor variables visible to that process are assumed to be given new values that are consistent with the monitor invariant. Thus, the monitor invariant must be strong enough to contain all the necessary information about the values the monitor variables will contain whenever the monitor is entered or left.

The restrictions necessary to make this simplified model of multiprogramming work are enforced by the Verifier during the preverification checking phase. A style of programming in which shared variables are accessed only by routines declared within monitors is required. This style of programming is strongly recommended by various structured programming enthusiasts, including Hoare [HOARE74] but it tends to result in programs with many tiny routines. Efficient implementation of such programs will require additional optimization support in the compiler.

2.1.4 Devices

In Pascal-F, devices have the syntax of variables, but very different semantics. The Verifier treats devices as procedures. Given the definitions

```
TYPE atod = DEVICE              (* A to D converter *)
                channel: 0..15; (* used to select channel *)
                data: 0..2047;  (* returns data value *)
            END;
```

the program fragment

```
atod.channel := 2;          (* select A/D channel *)
tab[2] := atod.data;        (* read A/D value *)
```

is treated by the Verifier much as if the program read

```
atodchannel(2);             (* select A/D channel *)
atoddata(tab[2]);           (* read A/D value *)
```

and **atodchannel** and **atoddata** were procedures. The Verifier makes no assumptions about the values returned by devices other than that values returned from devices are assumed to be in the range declared in the DEVICE declaration. Thus, DEVICE ranges should include the entire range of values that the device is electrically capable of producing. *The Verifier assumes that the hardware has been correctly described by the programmer.*

2.1.5 Enforcement

In this chapter are a number of restrictions on the way in which Pascal-F programs intended for verification may be written. The Verifier checks and enforces all of these restrictions before proceeding with verification. This step is referred to as preverification checking.

2.2 Extensions to Pascal-F for verification

2.2.1 Verification statements

The statements described here have no effect on the behavior of the program. They serve only as a documentation aid, to help the Verifier (and the human reader) understand the operation of the program.

Because verification statements are not allowed to call have any effect on the program, they cannot contain calls to functions that have side effects.

2.2.1.1 The ASSERT statement

```
ASSERT  ( <Boolean expression list> ) ;
```

This is the basic verification statement. An ASSERT statement is a claim that the given expressions are true whenever control reaches the statement. The Verifier will attempt to

prove this claim. Like all verification statements, the ASSERT statement has no effect on program execution.

2.2.1.2 The STATE statement

```
STATE  ( <Boolean expression list> ) ;
```

The STATE statement may only be written within the body of a loop, and represents the *loop invariant* of the loop. The Boolean expressions in the STATE statement must be true every time the STATE statement is reached. The Verifier uses the STATE statement as the place at which it will begin and end analysis of the loop. Each Boolean expression in the STATE statement is verified for the path around the loop and for the path into the loop. For the path around the loop, the path tracing starts at the STATE statement, goes around the loop once (backwards) and ends at the same STATE statement. Thus, the Boolean expression list must include all important information about variables that are changed by the loop body, but it need not mention variables that are left untouched. More advice on the use of the STATE statement appears in the section ‘‘What to do when a verification fails’’.

One (and only one) STATE statement must be written for every loop. The statement must contained in the loop body, at the top level. That is, if the loop body contains any compound statements then the STATE statement for the loop must appear outside those statements.

2.2.1.3 The MEASURE statement

```
MEASURE ( <numeric expression> ) ;
```

The MEASURE statement is used to prove that loops terminate. Every WHILE and REPEAT loop (except for deliberate infinite loops, which must begin with "WHILE true DO" or end with ‘‘UNTIL false;’’) must contain a MEASURE statement. The MEASURE statement must immediately follow the STATE statement associated with the loop. The <numeric expression> is a limit on the number of iterations of the loop body left to be performed. More specifically, the Verifier will insist that the value of the expression

- be greater than or equal to zero when control reaches the MEASURE statement, and
- decrease by at least one (for integers) or the precision of the expression (for fixed point numbers) each time control flows from the MEASURE statement around the loop back to the STATE statement.

How complex a MEASURE statement is required is a function of how obvious it is that the loop always terminates. FOR loops always terminate, and no MEASURE statement is used. If a loop uses a counter of one form or another, the MEASURE statement will

contain a simple expression involving the counter. For example, if the loop is counting i from 1 to 100, "MEASURE 100−i" would be a good choice. If the fact that a loop terminates is subtle, a complicated MEASURE statement, perhaps one involving EXTRA variables, will be necessary. If the loop does not terminate under some circumstances, no MEASURE statement, no matter how complex, will be accepted by the Verifier.

2.3 New expressions

The Verifier allows the use of some new constructs for building expressions. Some of these constructs can be used only in the special verification that will be described; others can be used in executable statements as well.

2.3.0.1 The DEFINED predicate

```
DEFINED(<variable>)

DEFINED(<array>,<low bound>,<high bound>)

DEFINED(<block name>)
```

DEFINED is a generic, built-in function that can be applied to any variable or part thereof and returns a Boolean value. DEFINED cannot be used in executable statements. A variable is said to be DEFINED whenever it is guaranteed to have a meaningful value. The following rules are used to determine whether an expression is DEFINED:

- All constants are DEFINED, including VALUE constants.
- Values obtained from DEVICE variables are DEFINED.
- A record variable is DEFINED if and only if all the fields of the record are DEFINED.
- An array variable is DEFINED if and only if all the entries of the array are DEFINED.
- A variable is DEFINED after it has been used on the left side of an assignment statement in which the right side was DEFINED.
- At the beginning of the body of a FOR loop, the index variable is DEFINED.
- A subscripted variable is DEFINED if the subscript is DEFINED, the subscript is in the range of the array bounds, and the specific array element being referenced is DEFINED.

Once a variable is DEFINED it remains DEFINED, unless an operation is performed that results in the variable receiving an indeterminate value. Only the following circumstances can result in a variable not being DEFINED:

- The variable is used in an index statement of a FOR loop is not DEFINED when the loop terminates.

- Immediately after the tag field of a variant record is changed, all the other fields in that record are no longer DEFINED.

The second form of DEFINED is used to test the definedness of portions of an array. The form

```
DEFINED(tab,i,j)
```

is true if the array **tab** has all elements DEFINED for elements with subscripts between **i** and **j** inclusive. This form is often used in loop invariants, inside the STATE statement.

The DEFINED predicate can also be applied to monitors and modules, as in the third form given above, but it has a slightly different meaning. If an INVARIANT has been declared for the monitor or module, the invariant is expected to hold whenever the construct is entered or left. There must be one exception to this rule, however. Since no variables are DEFINED when the program begins execution, the invariant cannot be expected to hold. It is the responsibility of the statements in the body of the monitor or module to initially establish the invariant. If m is a monitor or module, then DEFINED(m) is TRUE whenever m has been initialized. No routine exported from a monitor or module may be called unless DEFINED(m) is true.

2.3.0.2 The OLD annotation

At the beginning of every procedure and function, the Verifier implicitly saves the value at entry of every variable used as an input to the routine. In assertions, it is possible to refer to these values by placing an **.OLD** following a variable name or selector expression. Thus, if X is a parameter or variable, **X.OLD** denotes the value the parameter or variable had when the routine was entered. Old values of array variables are referenced with forms such as **X.[3].OLD** for an array reference, or **X.FIELDNAME.OLD** for field references. OLD may only be used within verification statements and PROOF statements, since this saving of OLD values does not occur in the running program.

The chief use of OLD is as a specification tool. For example, suppose one wanted to write a procedure to increment the value of a variable by one. The interface for this procedure might be

```
TYPE smallint = 1..100;
PROCEDURE bump(VAR x:  smallint);
ENTRY x < 100;
EXIT x = x.OLD + 1;
```

2.3.0.3 The IMPLIES operator

A new Boolean operator IMPLIES is added to the language. Informally, the expression p IMPLIES q means, "if p is TRUE, then q will be TRUE as well." If p and q are Boolean expressions, then p IMPLIES q has the value TRUE if p is FALSE or q is TRUE, and it has the value FALSE in all other cases. For purposes of precedence, IMPLIES is considered to be a relational operator.

The IMPLIES operator is for the most part used in verification statements, but it may be used in executable statements as well. Because it happens to be the case that FALSE <= TRUE, every IMPLIES operator in Pascal-F can be replaced by the operator <= without changing the meaning of the program. This practice is not advised, since the resulting program will be harder to read than the one using IMPLIES.

2.3.1 Verification declarations

To verify a program, more information about each routine is needed than what is necessary to compile a program. The Verifier expects routine headers that are more complex than those of standard Pascal. The extended syntax is:

```
<block> ::= <entry declaration part>
            <exit declaration part>
            <effect declaration part>
            <invariant declaration part>
            <depth declaration part>
            <constant declaration part>
            <type declaration part>
            <variable declaration part>
            <statement part>
```

Each declaration part is either empty, or consists of a keyword followed by a series of declarations, each of which is followed by a semicolon. The declarations that are not part of standard Pascal are explained in this section.

2.3.1.1 The ENTRY declaration section

```
ENTRY <Boolean expression series>;
```

The Boolean expressions given in this section are assertions that must be TRUE whenever the routine is called. The scope rules of Pascal-F prevent the Boolean expressions from containing variables other than parameters of the routine and variables global to the routine. In a routine exported from a module, ENTRY assertions cannot use variables local to the module. The purpose of the ENTRY section is to state restrictions on the values of these variables.

ENTRY assertions are a form of documentation as well as a verification requirement. Instead of writing "This procedure may only be called when filesopen is 0" in a comment, one writes

```
ENTRY filesopen = 0;
```

Some ENTRY assertions are automatically generated for each routine.

- For each value argument, an assertion is generated that the argument is DEFINED.
- For routines that are exported from a monitor or module m, an ENTRY assertion is generated consisting of the invariants for m.

Whenever a variable is given a value, the Verifier checks that the value is appropriate to the type of the variable. Therefore, any variable that is DEFINED has a value that is appropriate to its type. Therefore is not necessary to write ENTRY assertions that state that a parameter has a value appropriate for its subrange type.

When a variable **v** is passed to a **VAR** parameter **p** and used as an input variable, the Verifier automatically adds **DEFINED(p)** to the ENTRY conditions of the routine being called. Similarly, for a VAR output variable, **DEFINED(p)** would be added to the EXIT conditions. This automatic insertion of **DEFINED** predicates can be overridden by mentioning the formal argument involved in a user-supplied **DEFINED** in an ENTRY or EXIT condition. A common case in which the user must override the Verifier's automatic insertion is shown below.

```
PROCEDURE search(var n: integer);
ENTRY DEFINED(n) = DEFINED(n);
EXIT DEFINED(n);
BEGIN
    n := 0;
    WHILE (n < 100) AND (tab[n] <> 0) DO BEGIN
        STATE(defined(n), defined(tab));
        n := n + 1;
        END;
    END;
```

Here, we do not want to require that **n** be DEFINED at entry to *search*, but **n** appears to be an output variable since there are references to it within the procedure. The use of

```
DEFINED(n) = DEFINED(n)
```

prevents the Verifier from adding a requirement that **n** be DEFINED at input, but does not impose any requirement of its own.

There are rare cases in which a parameter p is only sometimes used for input depending on the values of the other parameters are. In this case, an ENTRY assertion of the form

```
C IMPLIES DEFINED(p)
```

should be used, where C is the condition under which the variable is read. Routines that have array or record VAR parameters may need complex ENTRY assertions indicating which portions of the array must be DEFINED at entry.

Monitors and modules may have ENTRY declaration sections. These refer to the initialization block of the monitor or module. For monitors and modules, the assertions in the ENTRY section must be true when the INIT statement for the monitor is executed,

and will be assumed true at the beginning of the initialization part of the block.

2.3.1.2 The EXIT declaration section

```
EXIT <Boolean expression series>;
```

The EXIT declarations define the state of the program upon exit from a routine by giving assertions that will be true at that time. The scope rules of Pascal-F limit the variables allowed in an ENTRY assertion to parameters and variables global to the routine. An additional restriction enforced is that value parameters cannot appear in an EXIT assertion, except as fields of the record **OLD.**

EXIT assertions tend to be long and complex, because everything that the routine does that is of importance to the caller must be described in the EXIT assertions. In some cases the EXIT assertions may be as long as the body of the routine.

If a routine has a VAR parameter p that is used for output, **DEFINED(p)** must be included among the EXIT assertions. If the parameter is only used for output some of the time, the situation must be described in an EXIT assertion of the form

```
C IMPLIES DEFINED(p)
```

where C is the condition under which p is used for output. If **p** is an array or record and only part of **p** is written, the situation must be described using more complex **EXIT** assertion.

2.3.1.3 The EFFECT declaration section

```
EFFECT <Boolean expression series>;
```

This declaration may only appear in a routine that is exported from a module. An EFFECT assertion is similar to an EXIT assertion, except that EFFECT assertions cannot contain references to variables local to the module.

Upon return from a ‘‘regular’’ routine (that is, one that is not part of a module) the Verifier uses the fact that the EXIT assertions (with actual arguments replacing formal parameters) for that routine are true. However, if the routine is exported from a module and called from outside the module, its EXIT assertions cannot be used since they may contain variables not visible to the caller. Instead, the EFFECT assertions are used by the Verifier. These assertions cannot contain variables local to the monitor. If a routine exported from a monitor is referenced from within the monitor, both EXIT and EFFECT assertions are used.

The following example shows how OLD, ENTRY, EXIT, and EFFECT are used to specify a module. The module also contains an INVARIANT declaration. INVARIANT declarations are explained in the next section.

```
MODULE stack;
EXPORTS size, push, pop;
VAR stp: 0..maxsize;
    buf: array [1..maxsize] of stackelt;

EFFECT size = 0;

INVARIANT FOR x: 1..maxsize
   ALL x<stp IMPLIES DEFINED(buf[x]);

PROOF FUNCTION size: 0..maxsize;
EXIT   size = stp;
BEGIN  size := stp END;

PROCEDURE push(x: stackelt);
ENTRY  size < maxsize;
EFFECT size = OLD.size + 1;
BEGIN  stp := stp + 1;
       buf[stp] := x
END;

FUNCTION pop: stackelt;
ENTRY  size > 0;
EFFECT size = OLD.size - 1;
BEGIN  pop := buf[stp];
       stp := stp - 1
END;

BEGIN stp := 0 END;
```

A few things should be noted about this example. The ENTRY assertions indicate under what conditions each routine can be called. They are written in terms of the proof function size. The EFFECT declarations describe what each routine does to size so that the users of the module can tell when it is valid to call **push** and **pop.** The EXIT assertion in size is used by the Verifier to translate the ENTRY and EFFECT assertions in **push** and **pop** into assertions about the variable **stp, which is (unbeknownst to the caller) changed by those routines.**

2.3.1.4 The INVARIANT declaration

```
INVARIANT <Boolean expression series>;
```

The assertions given in an INVARIANT declaration are required to hold at multiple points in a program.

- Putting an INVARIANT P in a routine is equivalent to putting ASSERT P statements at the beginning of the routine (i.e. as an ENTRY assertion), the end of the routine (as an EXIT assertion), before and after each WAIT statement in the routine, and before and after each call to another routine.
- Putting an INVARIANT P in a MONITOR is equivalent to putting ASSERT P statements at the end of the monitor block, before and after each WAIT statement in the monitor, before and after each call to a routine outside the monitor, and adding P as an INVARIANT assertion to each routine exported from the monitor. Note that an ASSERT P is not added to the beginning of the monitor block; the invariant is not assumed to hold until the monitor has been initialized.
- If the declaration INVARIANT P is put into a MODULE, the only variables on which P can depend are those local to the MODULE. The assertion P must be true at the end of the initialization block of the MODULE, before and after each WAIT statement in the module, before and after each call to a routine outside the module, and at the beginning and end of each routine exported from the MONITOR.

In other words, if an assertion P is declared to be an INVARIANT of a construct, P must be true whenever control passes into or out of that construct.

2.3.1.5 The DEPTH declaration

A recursive routine R1 is one that can call R2, which in turn calls R3, and so on until R1 is called again. A special case of recursion is a routine that calls itself. When a program uses recursion, it is necessary to prove that it is impossible to initiate an endless sequence of routine calls, none of which ever returns.

Every recursive routine must contain a DEPTH declaration of the form:

```
DEPTH <integer expression>;
```

By the scope restrictions of Pascal-F, the integer expression may only contain parameters of and variables global to the routine. Note that a fixed point expression cannot be used.

The DEPTH expression is an indication of how much time will be used by the routine. The ENTRY assertion for the routine must be strong enough to imply that the DEPTH expression is nonnegative whenever the routine is called.

From a recursive routine R1 two kinds of calls to other routines are possible. If R1 calls itself, or if it calls a routine R2 that can initiate a chain of routine calls that eventually leads to R1 being called again, the call is said to be a recursive call. Otherwise, the call is said to be nonrecursive.

Wherever a recursive routine R1 makes a recursive call to a routine R2 (a special case is when R1 and R2 are the same routine), it must be established that an infinite chain of calls is not being initiated. Let d1 be the value (at the time R1 was called) of the DEPTH expression declared for R1. Similarly, let d2 be the value (at the time R2 is being called) of the DEPTH expression declared for R2. The Verifier will attempt to prove that d2 is strictly less than d1. If this condition is proved for every recursive call, then each call in a chain must perform a successively easier task, so that every use of recursion must

eventually terminate.

2.3.2 EXTRA variables and PROOF statements

Quite often, it is not easy to demonstrate to the Verifier that a program works. One method of simplifying this task is to demonstrate that a program with some ‘debug code’ added works. For example, in proving that a linked list is correctly linked, it is easier to prove that a doubly-linked list is correct than that a singly-linked list is correct, because in the doubly-linked case, an invariant can be stated that claims that the operations of inserting and deleting from the list keep the backward and forward links consistent. Consistency for a singly-linked list is harder to define.

In a case such as the above, implementing a doubly-linked list may not be necessary for the operation of the program, but may be desirable for verification. EXTRA variables, for use in assertions, and PROOF statements, for manipulating EXTRA variables, are useful in such situations.

2.3.2.1 The EXTRA variable attribute

```
<variable name list>: EXTRA <type>;
```

The EXTRA attribute may be used in VAR declarations, in formal parameter declarations, and in RECORD definitions. The type of the result of a function may not have the EXTRA attribute. In each case, the attribute denotes a variable, parameter, or record field that is to be used for proof purposes only. That is, EXTRA variables must be used in such a way that all the EXTRA information can be removed from a program without affecting its execution. See the section on PROOF statements for more details.

2.3.2.2 The EXTRA function and procedure attribute

```
EXTRA FUNCTION <function definition>

EXTRA PROCEDURE <procedure definition>
```

Functions and procedures designated as EXTRA are solely for use in PROOF statements and assertions. Every parameter to an EXTRA routine is implicitly an EXTRA variable, and every statement in the routine is implicitly a PROOF statement.

2.3.2.3 RULE functions

```
RULE FUNCTION <function definition> BEGIN END;
```

Functions may be declared as RULE functions for use in proof rules. Such function definitions have no body.

Rule functions are used when an expression is needed in an assertion but the needed expression cannot be written as a simple Pascal-F expression. The use of rule functions is covered in detail in the chapter on rules.

Rule functions are restricted to results of types **integer, char,** and **boolean.** Arguments to rule functions may be declared as any valid Pascal-F type.

2.3.2.4 The PROOF statement

```
PROOF <statement>
```

Any executable statement, even a compound statement, may be turned into a PROOF statement by preceding the statement with the keyword PROOF. Such statements are ignored by the compiler. PROOF statements are used to manipulate EXTRA variables and to control conditional execution of other PROOF statements. The removal of all PROOF statements and EXTRA variables from a program must not change the execution of the program. The Verifier checks this restriction by enforcing rules that prevent any PROOF statement from affecting any non-PROOF variable.

Any expression containing an EXTRA variable will be referred to as a ‘proof expression’. The specific restrictions imposed to prevent PROOF statements and expressions from affecting program execution are as follows.

- Proof expressions may not contain calls to non-EXTRA functions that have side effects (other than the modification of EXTRA variables).
- Proof expressions can only be used in PROOF statements and as arguments to routines. The following parameter matching rules must be observed when calling non-EXTRA routines:
    - If a formal value parameter of a routine is an EXTRA variable, the corresponding argument cannot have any side effects (other than the modification of EXTRA variables).
    - If a formal VAR parameter of a routine is an EXTRA variable, the corresponding argument must also be an EXTRA variable.
    - If a formal parameter is not an EXTRA variable, the corresponding argument must not contain any EXTRA variables.
- EXTRA functions and procedures may be called only from within PROOF statements.
- Non-EXTRA variables may not be modified in any way by PROOF statements. This restriction applies to assignment statements, FOR loops, and routine calls.
- The multiprogramming statements WAIT, SEND, and INIT are forbidden in PROOF statements.

Study of the sample engine control program in Chapter 4, which contains a number of EXTRA variables, will give some insight into the use of this language feature.

2.4 Restrictions on Pascal-F programs

The restrictions given here are imposed to make the task performed by the Verifier easier. In most cases, the restrictions are in line with good programming practice. However, the concern here is not style but verifiability. A construct is prohibited only when there is some specific problem in handling that construct.

2.4.1 Restrictions on program structure

- The only legitimate ways to write non-terminating loops for a process is

```
WHILE true DO BEGIN <loop body> END;
```

or

```
REPEAT <loop body> UNTIL true;
```

  This construct may not appear within any other control structure. This restriction is imposed so that the Verifier can discriminate between accidental and deliberate non-terminating loops.

- The iteration variable of a FOR loop may not be modified from within the loop. Further, the value of the iteration variable is not DEFINED after the loop terminates. The Verifier issues an error message whenever a variable that is not DEFINED is accessed.

2.4.2 Restrictions on variant records

Variant records are not permitted by the verifier.

2.4.3 Restrictions on exception handling

Pascal-F incorporates a powerful exception handling mechanism similar to that in Ada. This version of the Verifier operates on the assumption that exception handlers are not used in normal operation and enforces the following rules.

- The RAISE statement, for exception handling, is considered to be a mechanism for recovering from hardware errors only. The Verifier will try to prove that no RAISE statement is ever executed.
- Exception handling routines are not examined.

2.4.4 Aliasing and side effects

Aliasing and side effects are two related phenomena that make programs difficult to understand and verify. Aliasing is the condition in which two names refer to the same variable. A side effect occurs when a function modifies one of its parameters or a global variable. The Verifier enforces restrictions that prevent variables from being changed in a fashion that it cannot detect, or in such a fashion that the resulting value of the modified variable depends of the order of evaluation.

The following program contains an example of aliasing.

```
PROGRAM pr;
VAR x: 1..100;

PROCEDURE p(VAR a: 1..100);
BEGIN
    a := 1;
    x := 2;
    ASSERT(a = 1);
END;
BEGIN
    p(x);
END pr.
```

Procedure p, viewed in isolation, is valid. The variable ‘a’ is clearly 1 after the body of the procedure has been executed. However, the call ‘p(x)’ causes the variables ‘a’ and ‘x’ to refer to the same variable within the procedure ‘p’. The assignment to ‘x’ will therefore change ‘a’ at the same time, and the assertion ‘a = 1’ will not be valid.

Because this sort of thing is far more often a cause of error than a useful feature of the language, it is prohibited.

When a function is called, VAR parameters of the function or variables global to the function can be modified. The modification of a variable by a function is called a "side effect" because the modification is usually not the primary purpose of the statement in which the function is called. Generally, programming with side effects is a dangerous practice because it is easy to forget that the side effect will take place.

This danger is diminished when using the Verifier, since the Verifier will detect side effects. However, there is a class of side effects whose result depends on the order of evaluation in expressions. As an example, suppose the functions f and g set their arguments to 1 and 2, respectively. The value given to x by the statement:

```
y := f(x) + g(x)
```

depends on whether the call to f or g is evaluated first. The Verifier allows statements to have side effects only if it can determine that the results of the side effects are independent of the order of evaluation in expressions.

The programmer can use the following simple rules to avoid problems with aliasing and side effects.

1. Do not pass the same variable to two different VAR parameters of a routine.
2. Do not pass a variable or any component thereof to a VAR parameter of a routine that sets or uses the variable as a global.
3. Within a function, do not modify VAR parameters or global variables, perform WAIT statements, or call a routine exported from a monitor, unless the function is called only in simple assignment statements.

4. Do not use functions exported from monitors in expressions; use them only as the right side of simple assignment statements.

The Verifier detects any violations of these rules.

2.4.5 Restrictions on multiprogramming

The restrictions required to make programs with multiple processes verifiable are somewhat severe. Unlike the other restrictions, which generally prohibit only language forms of little if any real use, the multiprogramming restrictions can be difficult to live with. The object of these restrictions, as discussed earlier in this chapter, is to allow the Verifier (and the programmer) to generally ignore the fact that semi-concurrent operations are taking place.

The major restriction required to make multiprogramming work is to require that any code referencing a static variable must be at the same priority as the variable. In Pascal-F, this is easy to enforce, because both the priority of both code and static variables is determined strictly by the priority of the module in which they are enclosed. The rules are as follows.

- No variable may be imported or exported from a monitor.
- A routine that has been exported from the monitor may not be called before the monitor has been initialized with an INIT statement.
- A monitor variable may not be passed as a VAR argument to a routine outside the monitor. (There are no restrictions on arguments passed by value.)
- A signal may not be a component of any other type.

2.5 Using Pascal-F with verification statements

2.5.1 Guidelines for writing verifiable programs

- Every variable should be of the minimum subrange type required for the range of values needed. For example, if negative values are illegal for a variable, its type should not permit negative values. Likewise, if zero is also prohibited, a positive range should be specified. These declarations give the Verifier extra information to use without extra writing by the programmer. They also can save space in the object program.
- Where alternatives are mutually exclusive, use the IF - THEN - ELSE structure in preference to consecutive IF statements so that it is clearly impossible for more than one alternative to be executed. Properly structuring a program holds down the number of possible paths to be traced out. Large numbers of paths make for long sessions with the Verifier.
- Use many short assertions instead of a few big ones. Do not lump assertions together with AND operators. The diagnostic messages, which refer to individual assertions, will then be more useful.

- Because many diagnostic messages are line-number oriented, it helps to put only one assertion on a source line.
- When writing loops, remember to use a STATE statement to describe what the state is after each iteration. Keeping the loop simple eases the difficult task of writing loop state assertions.
- The Verifier can make better deductions about addition and subtraction than it can about multiplication and division. It will be quite difficult to verify anything that depends on more than the most obvious properties of multiplication and division. Multiplication by constants is not a problem. The Verifier can deduce the possible range of the result of a division, but little else.
- Avoid scaling fixed-point operations so that truncation occurs. The Verifier can deduce the possible range of the result of a truncation, but little else.
- When progressing through an array with a loop, use a FOR loop unless early exit from the loop is planned. The Verifier supplies the proofs that FOR loops terminate.
- When processes at different priorities must communicate, one process will have to call a routine in another module to access any shared variables. This restriction is an incentive to reduce access to shared variables.
- Do not combine data that logically belongs to different priority code in the same record. Doing so will force the use of access routines unnecessarily.

2.5.2 Obtaining the most from the Verifier

Having a Verifier around encourages ‘‘defensive programming’’. Good programmers often write error detection into programs. Unfortunately, when program space or time are at a premium, it is not possible to put in (or in some cases leave in) all the traps for software bugs that should be there. When these traps are written as verifiable assertions, the checking can be done during verification, and there is no execution penalty.

In this sense, the most important defense against errors is extensive use of ENTRY assertions. Use ENTRY assertions as a documentation aid, to explain each routine to its users. These assertions (and module INVARIANT assertions) can be used to protect the routine from its callers, since the Verifier requires that the state of the system is what it is supposed to be when the routine is called.

Once all these ENTRY assertions are verified, they should be left in the program as a guide for those who must maintain the program. Over the long haul, one of the most important benefits of verification is that it allows the designer of the original program to leave behind rules about how the program is supposed to work. These rules might be unknown, forgotten, or ignored by future maintenance programmers, but if they are in the program text, the Verifier will use them.

3. Using the verifier

The Verifier runs on Digital Equipment Corporation VAX computers under Berkeley UNIX or Wollongong Eunice/VMS, or on SUN workstations. This chapter contains

instructions for using the verifier.

3.1 Invoking the Verifier

The Verifier is called using the UNIX command line:

```
pasver  [flags] <file>
```

This command initiates verification of the Pascal-F program in the named file. The file name must end in **.pf** indicating that the file is Pascal-F source.

If the **-dvcg** flag is given, messages will be printed indicating the progress of the verification, and failed verification conditions will be stored for examination by the user. The **-d** flag enables all internal debugging output, and should be used when trouble reports are submitted.

The verifier creates, in the current directory, a new directory for its scratch and history files. This new directory has the name of the program being verified, except that the trailing **.pf** is replaced with **_d** indicating a directory. The files in this directory are used to speed up reverifications when not all the program units of the program have been changed. The rules associated with the verification are also stored in this directory. Reverification is omitted for previously-verified program units when the unit is unchanged and the program unit was successfully verified in the past. Reverifications are much faster than original verifications.

3.2 Understanding error messages

The verifier generates error messages during three phases of processing; syntax checking, preverification checking, and verification. Samples appear below. Detection of an error during any phase prevents further phases from taking place, so only one of the three kinds shown below will appear as output from any given verification attempt.

3.2.1 Syntax error messages

```
prog.pf
     4.             IF x < 0 THEN x := 0)
 ****  14                                ^
  14:  ';' expected
Compilation complete -          1 errors detected
  *** Pass 2 deleted ***
Pass 1 error abort.
```

Syntax checking in the verifier is essentially the same as the Pascal-F compiler. This is to be expected, since the first passes of both are the same. The messages here are of the same types one would expect from a compiler. All messages here indicate errors; there are no ambiguities.

3.2.2 Preverification error messages

```
Pass 1:
Pass 2:
xxx.pf:
   9.      proc1(global1);
*** Variable "global1" is already used globally by "proc1". ***
1 error.
Pass 2 error abort.
```

Preverification checking is performed only after the entire program has been syntax checked, and information about procedures and global objects has been collected. This phase is primarily a check for inconsistencies between definition and use of objects. In addition, the restrictions necessary to make verification possible are enforced by this phase, and some common errors which can be caught efficiently in this phase are diagnosed. In the example above, there is an aliasing error; the variable *global1* is being passed as a **VAR** argument to a procedure which uses *global1* as a global variable. This is of course forbidden, since *proc1* will not behave as it normally would when a global and formal variable actually refer to the same location in memory.

One line of the source program is printed with each message to reduce the need to refer to a printed listing.

3.2.3 Verification error messages

In general, a message from this phase indicates that the possibility of a problem exists. As discussed earlier, a change in the program or the assertions will be required to eliminate the message.

Each message represents a proof failure along some path between a previous control point (procedure entry, loop invariant, wait, or STATE assertion) and the line displayed. Where more than one path exists, because of conditional statements, the path being traced out is described. This is done by stating the choice made at each conditional statement on the path.

```
Pass 1:
Pass 2:
Pass 3:

Verifying example6
Could not prove {example6.pf:18} table1[(j - 1) + 1] = 0
        (ASSERT assertion)
for path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} FOR loop exit

Could not prove {example6.pf:14} allzero(table1,1,i)
        (STATE assertion)
for path:
    {example6.pf:11} Start of "example6"
    {example6.pf:15} Back to top of FOR loop

Could not prove {example6.pf:14} allzero(table1,1,i)
        (STATE assertion)
for path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} Enter FOR loop

Could not prove {example6.pf:13} allzero(table1,1,i - 1)
        (ASSERT assertion)
for path:
    {example6.pf:11} Start of "example6"
    {example6.pf:15} Back to top of FOR loop

Could not prove {example6.pf:13} allzero(table1,1,i - 1)
    (ASSERT assertion)
for path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} Enter FOR loop

5 errors detected
```

The Verifier shows which specific assertion it could not prove, and for what path through the program proof was unsuccessful. A listing of the Pascal-F program to which the above messages refer appears in a later chapter.

3.3 How to proceed when a verification fails

The first attempt at a verification will produce many error messages. An orderly approach to dealing with these will be helpful. The messages may be divided into several classes, as shown below. Each class should be eliminated in order. When new errors appear in a class previously eliminated, the new errors should be dealt with before continuing work on the old.

3.3.1 Eliminate the syntax errors

First, if any syntax errors or preverification error messages are present, they must be eliminated before the verifier will attempt the verification phase. This is straightforward and the messages are usually unambiguous.

3.3.2 Eliminate any definedness problems

When verification-phase messages appear, the first thing to do is to look at all messages associated with definedness. These look like

```
{prog1.pf: 25} Cannot prove "x" is defined.
```

Messages like this indicate that the Verifier could not prove that a variable was initialized at some point where the value of the variable was used. All errors related to definedness should be eliminated before working on further problems. Often this will require adding definedness assertions such as

```
ENTRY DEFINED(x);
```

to procedure and function definitions. Adding an ENTRY condition such as the one above will usually eliminate the error message for the routine to which it is added, but since it places a new requirement on every caller to the routine the next verification attempt may well have new error messages concerning the callers of the routine. One works outward until the main program is reached.

The user should be aware that the Verifier generates ENTRY and EXIT definedness assertions internally for each procedure for each variable referenced (for ENTRY) and set (for EXIT) in the procedure but not mentioned by the user in the ENTRY and EXIT assertions. This convenience feature handles most common cases, but can be overridden by the user when required by mentioning the variable in a *DEFINED* clause. This mechanism usually does the right thing for simple variables. For more complex variables not fully initialized for all calls to the routine, the user will have to provide entry conditions of his own. If, for example, at entry to a routine, the array *tab* is only expected to be initialized from 1 to *x*, one would write an entry assertion of the form

```
ENTRY DEFINED(tab,1,x);
```

A special case is the assignment before use of a global variable or **var** argument to a procedure or function. For example, in

```
procedure p(var x: integer; y: integer);
BEGIN
    x := 1;
    IF y > 0 THEN x := x + 1;
END;
```

the formal parameter *x* seems to be an input and an output variable, since it is both set and used within *p*. Here, an entry condition of the form

```
ENTRY DEFINED(x) = DEFINED(x);
```

is required. This form is essentially meaningless but turns off the built-in assumption that *x* had to be DEFINED at any call to *p*.

3.3.3 Eliminate the run-time safety errors

Messages referring to array bounds and variable ranges should be addressed next. Again, it may be necessary to add ENTRY and EXIT assertions to do this. It may also be necessary to add terms to STATE invariants.

3.3.4 Eliminate ENTRY errors

When an error message associated with an ENTRY condition appears, the message will specify which call to the routine is causing the problem. First check the ENTRY statement to make sure that the requirement is what you had in mind; if so, the caller may need work.

3.3.5 Eliminate INVARIANT and EXIT errors

These refer to the state at the end of a routine.

3.3.6 Work on loop invariants

This is the really hard job in verification. Fortunately, when one is only trying to prove absence of fatal run-time errors, it is not too tough. A few simple cases cover most situations, of which the following is typical.

```
WHILE parens > 0 DO BEGIN
    printchar(')');
    parens := parens - 1;
    STATE(DEFINED(parens));
    END;
```

The invariant can go anywhere in the loop but usually placing it at the end of the loop is more convenient, as in the example below.

```
FOR i := 1 TO 100 DO BEGIN
    tab[i] := 0;
    STATE(DEFINED(tab,1,i));
    END;
```

When initializing an array of records, it is usually desirable to write a procedure which initializes one record in the array through a **var** argument, and use that procedure in the initialization loop. The record initializing procedure should have as its exit condition that the entire **var** argument is DEFINED.

3.3.7 Find all hard-to-prove assertions

Examine the remaining error messages. Look at each one and ask yourself ‘‘can you convince yourself informally that the assertion is true for that path at that point, purely by tracing backward along the indicated path and looking at the statements there?’’. If not, the program or the assertions need work. If so, defer working on that assertion until all the assertions that need work have been dealt with. Once all the easy assertions are out of the way, it is time to deal with the hard ones.

The Verifier can prove, without assistance, assertions that are true because of properties of addition, subtraction, multiplication by explicit constants, the relational operators, the Boolean connectives, and storing into and referencing arrays and records. This takes care of about 90-95% of all verification conditions. Beyond this point, the Verifier needs help.

Help is provided by adding ASSERT statements to the program and by adding *rules* to the rule database. Whenever an ASSERT statement is placed in a Pascal-F program, the Verifier will try to prove that it holds. For any statement after the ASSERT statement in the program, the assertion will be assumed true. Hard assertions should be preceded by an easy assertion or assertions (ones that the Verifier can prove) which imply the hard assertion by some formal argument the Verifier doesn’t yet know about. This argument should be something that depends only on the ASSERT statement and the following hard assertion. It will then be necessary to prove that argument as a rule. Proving rules is done with the Rule Builder as a separate job; when working on the program, all the hard assertions should be found and preceded with an easy assertion, until the entire verification is a success except for the errors from hard assertion preceded by easy ones. Then it is time to go to the Rule Builder, and probably to the resident Rule Builder expert.

An example is indicated.

```
FOR i := 1 TO 100 DO BEGIN
    table1[i] := 0;
    assert(table1[i] = 0);
    assert(allzero(table1,1,i-1));
    STATE(allzero(table1,1,i));
    END;
```

In this example, the STATE assertion is hard to prove. But if we add the two ASSERT

statements, both of which the Verifier can prove without much trouble, we could certainly argue that this makes it obvious that the STATE assertion is sound. If we had a rule that said

```
   a[i] = 0 and allzero(a,1,i-1)
implies
   allzero(a,1,i)
```

the Verifier would apply the rule and prove the assertion. So we now know exactly what rule we need, and can prove it with the Rule Builder.

3.3.8 Completing the debugging

With this sequential approach to debugging a verification, the rather forbidding prospect of eliminating all those error messages is somewhat less intimidating. Note, though, that no sound statement can be made about the program until *every* error message has been eliminated. One false assertion can cause any number of other problems to be hidden. As an example, writing

```
ENTRY false;
```

will eliminate all error messages for any routine, but will cause any caller of the routine to fail. (It is an interesting property of the Verifier that unreachable code need not work!). The same problem can be induced by accident; for example

```
ENTRY (x < 0) and (x > 100);
```

is essentially equivalent to the previous statement, since no number can satisfy both constraints. Thus, until *all* error messages have been eliminated, the verification is unsuccessful.

4. The Syntax of Pascal-F

This chapter describes the syntax of Pascal-F, including the verification statements. Lines marked with asterisks indicate productions altered to include the additional syntax necessary for verification. The syntax here is a superset of that given in the Pascal-F Language Reference Manual, and is given in the notation used in that manual.

4.1 Production Rules

<program> ::= <program heading> <outer block> .

<program heading> ::= PROGRAM <identifier> ;

<outer block> ::= <label declaration part>
<constant definition part>
<type definition part>
<value declaration part>
<variable declaration part>
<global routine declarations>
<statement part>

<label declaration part> ::=
<empty> | LABEL <label> {, <label>} ;

<entry declaration part> ::= <empty> | *
ENTRY <expression> {; <expression>} ; *

<exit declaration part> ::= <empty> | *
EXIT <expression> {; <expression>} ; *

<effect declaration part> ::= <empty> | *
EFFECT <expression> {; <expression>} ; *

<invariant declaration part> ::= <empty> | *
INVARIANT <expression> {; <expression>} ; *

<depth declaration part> ::= *
<empty> | DEPTH <expression> ; *

<constant definition part> ::= <empty> | CONST
<constant definition> {; <constant definition>} ;

<type definition part> ::= <empty> |
TYPE <type definition> {; <type definition>} ;

<variable declaration part> ::= <empty> |
VAR <variable declaration>
{; <variable declaration>} ;

<value declaration part> ::= <empty> |
VALUE <value declaration>
{; <value declaration> } ;

<global routine declarations> ::= { <global routine> ; }

```
<routine declaration part> ::= { <routine declaration> ;}

<statement part> ::= BEGIN <statement> {; <statement>}
                     {; <exception handler>}  END

<global routine> ::= <monitor declaration> |
                     <routine declaration>

<routine declaration> ::= <module declaration> |
                          <function declaration> |
                          <procedure declaration>

<module declaration> ::=
               <module heading> <block>

<module heading> ::=
               MODULE  <identifier> ;
                 <export list>  <import list>

<export list> ::= EXPORTS  <identifier> {, <identifier>} ;
                  |  <empty>

<import list> ::= IMPORTS  <identifier> {, <identifier>}  ;
                  |  <empty>

<monitor declaration> ::=
               <monitor heading> <block>

<monitor heading> ::= MONITOR <identifier> PRIORITY
               <priority level> ; <export list>  <import list>

<priority level> ::= <constant>

<procedure declaration> ::= <procedure heading> <block>

<procedure heading> ::= <possibly extra> PROCEDURE                    *
               <identifier> <parameters> ;                            *

<function declaration> ::= <function heading> <block>

<function heading> ::= <possibly extra> FUNCTION <identifier>         *
               <parameters> : <result type> ;                         *

<parameters> ::= ( <formal parameter section>
               {; <formal parameter section>} ) |  <empty>
```

```
<formal parameter section> ::= <parameter group> |
            VAR <parameter group>

<parameter group> ::= <identifier> {, <identifier>} :                    *
              <possibly extra> <type identifier>                         *

<block> ::= <label declaration part>
          <entry declaration part>                                       *
          <exit declaration part>                                        *
          <effect declaration part>                                      *
          <invariant declaration part>                                   *
          <depth declaration part>                                       *
          <constant definition part>
          <type definition part>
          <value declaration part>
          <variable declaration part>
          <routine declaration part>
          <statement part>

<value declaration> ::= <identifier> =
              <type identifier> ( <value list> )

<value list> ::= <value> { , <value> }

<value> ::= <constant> | ( <value list> )

<compound statement> ::= BEGIN <statement> {; <statement>} END

<statement> ::= <unlabeled statement> |                                  *
              <label> : <unlabeled statement> |                          *
              PROOF <unlabeled statement> |                              *
              <verification statement>                                   *

<verification statement> ::= <assert statement> |                        *
                        <summary statement>                              *

<unlabeled statement> ::= <simple statement> |
              <structured statement>

<simple statement> ::= <assignment statement> |
                    <procedure statement> |
                    <init statement> |
                    <send statement> |
                    <wait statement> |
                    <raise statement> |
                    <empty statement>
```

<structured statement> ::= <compound statement> |
<conditional statement> |
<repetitive statement> |
<with statement>

<exception handler> ::=
WHEN <exception spec> DO <statement>

<exception spec> ::= <exception name> {, <exception name>} |
OTHERS

<exception name> ::= <identifier>

<constant definition> ::= <identifier> = <constant>

<constant> ::= <unsigned constant> | <sign> <unsigned constant>

<type definition> ::= <identifier> = <type>

<type> ::= <simple type> | <structured type>

<simple type> ::= <ordinal type> |
<fixed point type> |
<signal type>

<ordinal type> ::= <enumerated type> |
<subrange type> |
<type identifier> |

<enumerated type> ::= (<identifier> {, <identifier>})

<subrange type> ::= <constant> .. <constant>

<type identifier> ::= <identifier>

<fixed point type> ::= FIXED <constant> .. <constant>
PRECISION <constant>

<signal type> ::= SIGNAL <hardware mapping>

<structured type> ::= <unpacked structured type> |
PACKED <unpacked structured type> |
DEVICE <hardware mapping> <field list> END

<hardware mapping> ::= [<address>] | <empty>

<address> ::= <expression>

<unpacked structured type> ::= <array type> |
<record type> |
<set type>

<array type> ::= ARRAY [<index type> {, <index type>}] OF
<component type>

<index type> ::= <ordinal type>

<component type> ::= <type>

<record type> ::= RECORD <field list> END

<field list> ::= <fixed part> |
<fixed part> ; <variant part> |
<variant part>

<fixed part> ::= <record section> {, <record section>}

<record section> ::= <field identifier> {, <field identifier>} *
: <possibly extra> <type> | <empty> *

<variant part> ::= CASE <tag field> <possibly extra> *
<type identifier> OF <variant> {; <variant>} *

<tag field> ::= <identifier> : *

<variant> ::= <case label list> : (<field list>) | <empty>

<set type> ::= SET OF <base type>

<base type> ::= <simple type>

<variable declaration> ::= <identifier> {, <identifier>} : *
<possibly extra> <type> *

<possibly extra> ::= <empty> | EXTRA *

<result type> ::= <type identifier>

<assignment statement> ::= <variable> := <expression> |
<function identifier> := <expression>

<variable> ::= <entire variable> {<component part>}

<component part> ::= [<expression> {, <expression>}] |
. <field identifier>

<entire variable> ::= <variable identifier> | <field identifier>

<variable identifier> ::= <identifier>

<field identifier> ::= <identifier>

<expression> ::= <relation> *

<relation> ::= <simple expression> <relation part> *

<relation part> ::= <empty> | <relational operator> <relation> *

<relational operator> ::= = | <> | '< | '<= | >= | *
> | IN | IMPLIES *

<simple expression> ::= <unsigned simple expression> |
<sign> <unsigned simple expression>

<sign> ::= + | -

<unsigned simple expression> ::= <term> <addend>

<addend> ::= <empty> |
<adding operator> <unsigned simple expression>

<adding operator> ::= + | - | OR

<term> ::= <factor> <multiplier part>

<multiplier part> ::= <empty> | <multiplying operator> <term>

<multiplying operator> ::= * | / | DIV | MOD | AND

<factor> ::= <variable> | <unsigned constant> | (<expression>) |
<function designator> | <set> | NOT <factor>

<unsigned constant> ::= <unsigned number> | <string> |
<constant identifier>

<constant identifier> ::= <identifier>

<function designator> ::= <function identifier>
        <actual parameter part>

<function identifier> ::= <identifier>

<set> ::= [<element list>]

<element list> ::= <element> {, <element>} | <empty>

<element> ::= <expression> <range part>

<range part> ::= <empty> | .. <expression>

<procedure statement> ::= <procedure identifier>
        <actual parameter part>

<procedure identifier> ::= <identifier>

<actual parameter part> ::= <empty> | (<actual parameter>
    {, <actual parameter>})

<actual parameter> ::= <expression>

<label> ::= <constant>

<empty statement> ::= <empty>

<conditional statement> ::= <if statement> | <case statement>

<if statement> ::= IF <expression> THEN <statement> <else part>

<else part> ::= <empty> | ELSE <statement>

<case statement> ::= CASE <expression> OF <case list element>
    {; <case list element>} END

<case list element> ::= <case label list> : <statement> |
    <empty>

<case label list> ::= <case label> {, <case label> }

<case label> ::= <constant>

<repetitive statement> ::= <while statement> |
                <repeat statement> |
                <for statement>

<while statement> ::= WHILE <expression> DO <loop body> *

<repeat statement> ::= REPEAT <loop body> *
UNTIL <expression>

<for statement> ::= FOR <control variable> := <for list> DO
<loop body> *

<for list> ::= <initial value> <direction> <final value>

<direction> ::= TO | DOWNTO

<control variable> ::= <identifier>

<initial value> ::= <expression>

<final value> ::= <expression>

<loop body> ::= BEGIN {<statement> ;} <state statement> *
<optional measure statement> {; <statement>} END *

<optional measure statement> ::= ; <measure statement> | <empty> *

<with statement> ::= WITH <record variable list> DO <statement>

<record variable list> ::= <record variable> {, <record variable>}

<record variable> ::= <identifier>

<init statement> ::= INIT <monitor identifier>

<monitor identifier> ::= <identifier>

<raise statement> ::= RAISE <exception name>

<send statement> ::= SEND <signal name>

<wait statement> ::= WAIT <signal name>

<assert statement> ::= ASSERT (<expression> {, <expression>}) *

<summary statement> ::= SUMMARY (<expression> {, <expression>}) *

<state statement> ::= STATE (<expression> {, <expression>}) *

<measure statement> ::= MEASURE (<expression>) *

<signal name> ::= <identifier>

5. Examples

The examples in this chapter are designed to aid the user in learning to use the Verifier. For the first example, a square root calculation, verification conditions have been generated by hand and informal proofs given. This example is intended to give the user some insight into how the Verifier examines programs.

The second example, a set of routines for managing a circular buffer, shows the use of the verifier to verify only absence of run-time errors, and the maintenance of a simple invariant. It is useful to note how few assertions were required. This example has been passed by the Verifier in the form shown.

The third example, a very simple-minded engine control program, is intended to illustrate the use of assertions and proof variables in constructing a verifiable Pascal-F program. It also illustrates means of programming within the restrictions required for a reliable and verifiable multi-process program. This example also has been passed by the Verifier.

5.1 A simple example worked by hand

In this example the reader is taken ‘behind the scenes’, so to speak, to see how program verification can be performed manually. The function shown here is a fixed point square root routine. In addition to the normal proof of runtime-error-free operation, a full proof of correctness is attempted, which is to say that we actually try to prove that the function computes square root to within some explicit error bound. This example is rather more complex mathematically than most parts of control programs, and as will be seen the verification conditions generated are sometimes difficult mathematically, especially when one bears in mind that the rules of fixed-point arithmetic are being interpreted strictly.

Figure 1. Function before addition of verification statements

```
(*
        Square root by bisection

        This technique has the useful property that it
        it will work for any monotonic function.
*)
FUNCTION sqrt(s: FIXED 2.0 .. 100.0 PRECISION 0.1):
               FIXED 1.0 .. 10.0 PRECISION 0.1;
VAR
    x, lowbound, highbound: FIXED 1.0 .. 100.0 PRECISION 0.1;
BEGIN
    x := s;                      (* set initial try *)
    lowbound := 1.0;             (* lowest possible square root *)
    highbound := x;              (* highest possible *)
                                 (* stop when interval tiny *)
    WHILE highbound - lowbound > 0.1 DO
    BEGIN                        (* choose new trial value *)
        x := (highbound + lowbound) / 2;
        IF x*x > s THEN BEGIN    (* if x is too big *)
            highbound := x;      (* then answer must be below x *)
        END ELSE BEGIN           (* if x is too small *)
            lowbound := x;       (* then answer must be above x *)
        END;
    END;
    sqrt := x;                   (* return answer *)
END;                             (* of sqrt *)
```

The first step in documenting the function for verification is to define what the procedure is supposed to do. This is simple in this case, because square root is easy to define. An exit assertion of the form

```
    EXIT abs(x * x - s) < 0.2;
```

describes the desired result.

The next step is to figure out how to prove that the function will produce the desired result. We can prove this by noting the following facts about the situation that exists when the loop exits.

- x is between lowbound and highbound
- s is between lowbound squared and highbound squared
- lowbound is less than or equal to highbound

- lowbound is within 0.1 of highbound

All these conditions except the last hold for every iteration of the loop, and thus form part of the loop invariant. These insights must be provided to the verifier in a STATE statement.

Proving that the loop terminates requires that we find some measure that decreases as the loop iterates. Since the algorithm operates by closing the interval between highbound and lowbound, the difference between these two is a suitable value to appear in the MEASURE statement.

Figure 2. Function after addition of verification statements

```
(*
        Square root by bisection

        This technique has the useful property that it
        it will work for any monotonic function.
*)
FUNCTION sqrt(s: FIXED 2.0 .. 100.0 PRECISION 0.1):
                FIXED 1.0 .. 10.0 PRECISION 0.1;
EXIT abs(sqrt*sqrt - s) <= 0.2;  (* definition of result *)

VAR
    x, lowbound, highbound: FIXED 1.0 .. 100.0 PRECISION 0.1;
BEGIN
    x := s;                         (* set initial try *)
    lowbound := 1.0;                (* lowest possible square root *)
    highbound := x;                 (* highest possible *)
                                    (* stop when interval tiny *)
    WHILE highbound - lowbound > 0.1 DO
    BEGIN
        MEASURE highbound - lowbound;
        STATE  lowbound*lowbound <= s,
               highbound*highbound >= s,
               highbound >= lowbound,
               x <= highbound,
               x >= lowbound;
                                    (* choose new trial value *)
        x := (highbound + lowbound) / 2;
        IF x*x > s THEN BEGIN       (* if x is too big *)
            highbound := x;         (* then answer must be below x *)
        END ELSE BEGIN              (* if x is too small *)
            lowbound := x;          (* then answer must be above x *)
        END;
    END;
    sqrt := x;                      (* return answer *)
END;                                (* of sqrt *)
```

We can now attempt actual verification. The first step in verification is to trace out all possible paths of control flow. Paths begin and end at the boundaries of procedures and at STATE statements. There are thus six paths in this function.

1. The path starting at the beginning of the function, treating the WHILE condition as true, and ending at the STATE statement.

2. The path starting at the beginning of the function, treating the WHILE condition as false, and ending at the end of the procedure.
3. The path starting at the STATE statement, treating the IF condition as true, going around the loop, treating the WHILE condition as true, and ending at the STATE statement.
4. The path starting at the STATE statement, treating the IF condition as false, going around the loop, treating the WHILE condition as true, and ending at the STATE statement.
5. The path starting at the STATE statement, treating the IF condition as true, going around the loop, treating the WHILE condition as false, and ending at the end of the procedure.
6. The path starting at the STATE statement, treating the IF condition as false, going around the loop, treating the WHILE condition as false, and ending at the end of the procedure.

After working out the control flow in the program, the next step is to locate all the assertions which must be verified for the path and generate a verification condition for each. Assertions come not only from the user's EXIT and STATE statements but from internal requirements needed to prevent run-time errors.

For the first path, it is necessary to verify all the following conditions.

1. A range error does not occur at 'x := s'. (No problem, the type of x has less restrictive bounds than that of s.)
2. A range error does not occur at 'lowbound := 1.0'. (No problem, the constant value is in the correct range.)
3. A range error does not occur at 'highbound := x'. (No problem, the types match.)
4. The value in the MEASURE statement is non-negative. (Since lowbound is 1.0, and highbound is constrained by its type to be 2.0 or greater, this requirement is met.) Note that we are only concerned with the first time through the loop on this path.
5. The STATE assertion 'lowbound * lowbound <= s' holds. (Since 1.0 * 1.0 <= 2.0, this holds.)
6. The STATE assertion 'highbound * highbound >= s' holds. (Since on this path highbound = x, we need only prove that x > 2.0 IMPLIES x*x > x.)
7. The STATE assertion 'highbound >= lowbound' holds. (We know that highbound is not less than than 2.0 and that lowbound is 1.0, so this is no problem.)
8. The STATE assertion 'x <= highbound' holds. (x is equal to highbound; no problem.)
9. The STATE assertion 'x >= lowbound' holds. (We know that lowbound is 1.0, and that x is 2.0 or greater, so this is no problem.)

Proceeding on to path two, the path on which the WHILE statement body is never

executed, we find an interesting phenomenon. The loop is entered under the following conditions:

```
lowbound = 1.0
highbound >= 2.0
```

For path two to be executed, the condition for exiting the loop,

```
NOT(highbound - lowbound > 0.1)
```

would have to be true at initial entrance to the loop. Since these three conditions cannot all be true simultaneously, this path is never executed, and the loop body is always executed at least once when the function is called. All verification conditions for this path are true since the requirement for entry is false. Note that there is no dead code; there merely is no case where a specific set of decisions are taken when passing through several conditional statements. It is not unusual for this to happen and this presents no problems.

We now reach the first of the two interesting paths, the one around the loop taking the true branch at the IF statement. This path starts and ends at the STATE statement.

The STATE statement defines the loop invariant. We have already proved (for path one) that the loop invariant is true at entry to the loop. We must now prove that if the loop invariant is true for a given iteration of the loop, it will still be true at the end of that iteration. By proving this, we prove by induction that the loop invariant is true for every iteration of the loop.

This path is complex enough that each verification condition is written out formally as a proposition to be proven.

1. The first verification condition is produced by trying to prove that the invariant condition

   ```
   lowbound * lowbound <= x
   ```

   holds. The verification condition generated is

```
      lowbound >= 1.0 AND lowbound <= 100.0
  AND highbound >= 1.0 AND highbound <= 100.0
  AND s >= 2.0 AND s <= 100.0
  AND lowbound * lowbound <= s
  AND highbound * highbound >= s
  AND highbound >= lowbound
  AND x <= highbound
  AND x >= lowbound
  AND xNEW = (highbound + lowbound) / 2
  AND xNEW * xNEW > s
  AND highboundNEW = xNEW
  AND highboundNEW - lowbound > 0.1
IMPLIES
  lowbound * lowbound <= s;
```

This simplifies to

```
  lowbound * lowbound <= s
IMPLIES
  lowbound * lowbound <= s;
```

which is obviously true. This is a trivial case, because lowbound did not change on this path.

2.

```
      lowbound >= 1.0 AND lowbound <= 100.0
  AND highbound >= 1.0 AND highbound <= 100.0
  AND s >= 2.0 AND s <= 100.0
  AND lowbound * lowbound <= s
  AND highbound * highbound >= s
  AND highbound >= lowbound
  AND x <= highbound
  AND x >= lowbound
  AND xNEW = (highbound + lowbound) / 2
  AND xNEW * xNEW > s
  AND highboundNEW = xNEW
  AND highboundNEW - lowbound > 0.1
IMPLIES
  highboundNEW*highboundNEW >= s;
```

which simplifies to

```
  xNEW * xNEW > s
IMPLIES
  xNEW * xNEW >= s
```

which is true.

3.

```
        lowbound >= 1.0 AND lowbound <= 100.0
    AND highbound >= 1.0 AND highbound <= 100.0
    AND s >= 2.0 AND s <= 100.0
    AND lowbound * lowbound <= s
    AND highbound * highbound >= s
    AND highbound >= lowbound
    AND x <= highbound
    AND x >= lowbound
    AND xNEW = (highbound + lowbound) / 2
    AND xNEW * xNEW > s
    AND highboundNEW = xNEW
    AND highboundNEW - lowbound > 0.1
  IMPLIES
    highboundNEW >= lowbound;
```

which simplifies to

```
        lowbound >= 1.0
    AND highbound >= 1.0
    AND highbound >= lowbound
  IMPLIES
    (highbound + lowbound) / 2 >= lowbound
```

which is true.

4.

```
        lowbound >= 1.0 AND lowbound <= 100.0
    AND highbound >= 1.0 AND highbound <= 100.0
    AND s >= 2.0 AND s <= 100.0
    AND lowbound * lowbound <= s
    AND highbound * highbound >= s
    AND highbound >= lowbound
    AND x <= highbound
    AND x >= lowbound
    AND xNEW = (highbound + lowbound) / 2
    AND xNEW * xNEW > s
    AND highboundNEW = xNEW
    AND highboundNEW - lowbound > 0.1
  IMPLIES
    xNEW <= highboundNEW;
```

which transitivity of equality shows to be true.

5. The possibility that overflow might occur in the statement

```
x := (highbound + lowbound) / 2;
```

must be considered. The verification condition

```
        lowbound >= 1.0 AND lowbound <= 100.0
    AND highbound >= 1.0 AND highbound <= 100.0
    AND s >= 2.0 AND s <= 100.0
    AND lowbound * lowbound <= s
    AND highbound * highbound >= s
    AND highbound >= lowbound
IMPLIES
        (highbound + lowbound) / 2 >= 1.0
    AND (highbound + lowbound) / 2 <= 100.0
```

describes this, and this can be shown to always hold based solely on the restrictions on lowbound and highbound.

6. It is necessary to prove that the loop terminates. This is done by showing that the value in the MEASURE statement decreases with each loop iteration but never becomes negative.

```
        lowbound >= 1.0 AND lowbound <= 100.0
    AND highbound >= 1.0 AND highbound <= 100.0
    AND s >= 2.0 AND s <= 100.0
    AND lowbound * lowbound <= s
    AND highbound * highbound >= s
    AND highbound >= lowbound
    AND x <= highbound
    AND x >= lowbound
    AND xNEW = (highbound + lowbound) / 2
    AND xNEW * xNEW > s
    AND highboundNEW = xNEW
    AND highboundNEW - lowbound > 0.1
IMPLIES
        (highboundNEW - lowbound) < (highbound - lowbound)
    AND (highboundNEW - lowbound >= 0);
```

This immediately reduces to

```
        lowbound >= 1.0 AND lowbound <= 100.0
    AND highbound >= 1.0 AND highbound <= 100.0
    AND highbound >= lowbound
    AND ((highbound + lowbound) / 2) - lowbound > 0.1
  IMPLIES
    AND (highbound + lowbound) / 2 < highbound;
        (highbound + lowbound) / 2 >= lowbound;
```

which is true.

The next path to be considered is the same as the one above, except that the IF branch takes the 'false' path. We will spare the reader the details of this path, which are similar to those shown above.

The last two paths start at the STATE statement and go to the top of the loop, but exit at the WHILE statement rather than continuing in the loop, finally ending at the EXIT assertion. Let us first consider the path through the 'true' branch.

1. The replacement

```
sqrt := x;          (* return answer *)
```

implies a restriction on x that x is less than or equal to 10.0, because the type of the function 'sqrt' is FIXED 1.0 .. 10.0, while the type of 'x' is FIXED 1.0 .. 100.0, leading to the verification condition

```
        lowbound >= 1.0 AND lowbound <= 100.0
    AND highbound >= 1.0 AND highbound <= 100.0
    AND s >= 2.0 AND s <= 100.0
    AND lowbound * lowbound <= s
    AND highbound * highbound >= s
    AND highbound >= lowbound
    AND x <= highbound
    AND x >= lowbound
    AND xNEW = (highbound + lowbound) / 2
    AND xNEW * xNEW > s
    AND highboundNEW = xNEW
    AND highboundNEW - lowbound <= 0.1
  IMPLIES
    xNEW <= 10.0;
```

Performing obvious simplifications, we obtain

```
      lowbound <= 100.0
  AND highbound <= 100.0
  AND lowbound * lowbound <= 100.0
  AND highbound * highbound >= 2.0
  AND highbound >= lowbound
  AND xNEW = (highbound + lowbound) / 2
  AND xNEW - lowbound <= 0.1
IMPLIES
  xNEW <= 10.0;
```

which we fail to prove.

Let us see why. A counterexample to the above verification condition is

```
lowbound = 10.0
highbound = 10.2
xNEW = 10.1
```

which would cause overflow. There are two possibilities to be considered; either the Verifier does not have enough correct information to constrain the value more, because of an ill-chosen STATE assertion, or the program contains a bug.

In this case, the program indeed contains a bug; an attempt to take the square root of 100.0 will compute a value of 10.1, which is within the desired error tolerance for the square root routine but not within the range of approved values for the function. This is a good example of the sort of bug the Verifier is good at finding but which might be overlooked without verification.

2. Continuing onward, our last verification condition for this path is that the square root function gets the right answer, or

```
      lowbound >= 1.0 AND lowbound <= 100.0
  AND highbound >= 1.0 AND highbound <= 100.0
  AND s >= 2.0 AND s <= 100.0
  AND lowbound * lowbound <= s
  AND highbound * highbound >= s
  AND highbound >= lowbound
  AND x <= highbound
  AND x >= lowbound
  AND xNEW = (highbound + lowbound) / 2
  AND xNEW * xNEW > s
  AND highboundNEW = xNEW
  AND highboundNEW - lowbound <= 0.1
IMPLIES
  abs(xNEW * xNEW - s) <= 0.2;
```

Opening up the definition of ‘abs’ gives us

```
      lowbound >= 1.0 AND lowbound <= 100.0
  AND highbound >= 1.0 AND highbound <= 100.0
  AND s >= 2.0 AND s <= 100.0
  AND lowbound * lowbound <= s
  AND highbound * highbound >= s
  AND highbound >= lowbound
  AND x <= highbound
  AND x >= lowbound
  AND xNEW = (highbound + lowbound) / 2
  AND xNEW * xNEW > s
  AND highboundNEW = xNEW
  AND highboundNEW - lowbound <= 0.1
IMPLIES
     ((xNEW * xNEW > s) AND  (xNEW * xNEW - s) < 0.2))
  OR ((xNEW * xNEW < s) AND  (s - xNEW * xNEW) < 0.2));
```

which simplifies to

```
      lowbound >= 1.0 AND lowbound <= 100.0
  AND highbound >= 1.0 AND highbound <= 100.0
  AND s >= 2.0 AND s <= 100.0
  AND lowbound * lowbound <= s
  AND highbound * highbound >= s
  AND highbound >= lowbound
  AND xNEW = (highbound + lowbound) / 2
  AND xNEW * xNEW > s
  AND xNEW - lowbound <= 0.1
IMPLIES
  xNEW * xNEW - s < 0.2;
```

which if augmented by the previously proved information that xNEW is greater than or equal to lowbound, can, with some difficulty, proved true.

This completes the analysis of the example. One clear bug was found, as mentioned above; an attempt to compute the square root of 100 will result in an out-of-range result. This sort of boundary condition bug is common, and is one of the things which the Verifier is good at finding.

5.2 Circular buffering routines

This simple example shows a common program component; a set of circular buffering routines. Here the Verifier has been used to show absence of run-time errors and the maintenance of a ‘‘sanity invariant’’ which if violated would indicate that the various pointers were out of synchronism.

The main program here is a dummy one; the object here is to verify the subroutines.

```
program circle;
{
        Circular Buffering Module        Version 1.9 of 1/5/83
}
monitor circlebuf priority 5;
exports bufget, bufput;
const bufsize = 20;
type  bufindex = 1..20;                 { position in buffer }
      bufarray = array [bufindex] of char;
      buffer = record                   { buffer structure }
        bufin: bufindex;                { next position to insert }
        bufout: bufindex;               { next position to read }
        bufcount: 0..bufsize;           { chars in buffer }
        buf: bufarray;                  { the buffer itself }
        end;
var b: buffer;                          { the buffer }
invariant defined(b);                   { the buffer is always defined }
                                        { buffer sanity }
        ((b.bufout + b.bufcount) = b.bufin)
        or
        ((b.bufout + b.bufcount - bufsize) = b.bufin);
{
        bufput  --  put in buffer
}
function bufput(ch: char)               { char to insert }
                : boolean;              { returns true if insert OK }
begin
    if b.bufcount < bufsize then begin  { if buffer not full }
        b.bufcount := b.bufcount + 1;   { increment buffer count }
        b.buf[b.bufin] := ch;           { store char in buffer }
        assert(defined(b.buf,1,bufsize));{ array still defined }
        if b.bufin = bufsize then       { if at max }
            b.bufin := 1                { reset to start }
        else b.bufin := b.bufin + 1;    { otherwise increment }
        bufput := true;                 { success }
    end else begin                      { if full }
        bufput := false;                { insert fails }
        end;
end {bufput};
```

```
{
        bufget  --  get from buffer
}
function bufget(var ch: char)           { char returned }
        : boolean;                      { true if successful }
exit return implies defined(ch);        { char only if not empty }
begin
    if b.bufcount > 0 then begin        { if buffer not empty }
        b.bufcount := b.bufcount - 1;   { decrement buffer count }
        assert(defined(b));             { still all defined }
        ch := b.buf[b.bufout];          { get char from buffer }
        if b.bufout = bufsize then      { if at max }
            b.bufout := 1               { reset to start }
        else b.bufout := b.bufout + 1;  { otherwise increment }
        bufget := true;                 { success }
    end else bufget := false;           { fails if empty }
end {bufget};
```

```
{
        buffer initialization block
}
var i: bufindex;
begin
    for i := 1 to 20 do begin           { clear to spaces }
        b.buf[i] := ' ';
        assert(defined(b.buf,1,i-1));   { still defined up to i-1 }
        state(defined(i),
              defined(b.buf,1,i));
        end;
    b.bufout := 1;                      { start at 1 }
    b.bufin := 1;                       { end at 1 }
    b.bufcount := 0;                    { length 0 }
end {circlebuf};
var stat: boolean;                      { status from routines above }
    ch: char;                           { working char }
begin {main}
    init circlebuf;
    stat := bufput('x');
    stat := bufget(ch);                 { get a char }
    if stat then begin                  { if we got a char }
        end;
end.
```

The Verifier can verify these routines in about four minutes. This verification does require rules, but all the rules needed are in the Rule Builder's standard database.

```
Verifying circle
No errors detected

Verifying circlebuf
No errors detected

Verifying circlebuf-bufget
No errors detected

Verifying circlebuf-bufput
No errors detected
```

5.3 An example in the form of an engine control program

The ‘‘engine control’’ program here is not related to any real engine or control electronics. It is an example of a way in which a real-time program might be written. The program was written primarily to illustrate the kinds of things one might attempt to prove about a program of this type. It also demonstrates that it is not overly difficult to program under the restriction that monitors may not import or export variables. The program was written two years before the Verifier was operational, and appeared in a preliminary version of this manual.

This program has been verified by the Verifier. Quite a number of bugs were found in the program during the process. Most of the work in getting the Verifier to accept the program was in discovering the invariants needed for the *excluder* module. Most of the bugs found were related to proper handling of failure of the clock or crankshaft interrupts. Note that the invariants provided to check proper engine operation now hold even if the crankshaft interrupt or clock interrupt never comes. For example, there is code to fire the spark (belatedly) if the next crankshaft interrupt comes in before the spark has been fired under normal timing rules. This code was required before the constraint of one-spark-per-crank-pulse could be met.

```
program simpleengine;                   (*      version 1.50 of 1/14/83
(*
        sample engine control program

   This is a sample program written to illustrate some features of
Pascal-F as extended for verification purposes.  The program has a
rather simple-minded model of the engine, and controls only the
fuel pump and spark.  The only inputs available to the program
are the clock, and the shaft position pulse.
   This program does not interface with any existing engine hardware.

                                        John Nagle
                                        Ford Aerospace and
                                        Communications Corporation
                                        Western Development Labs
*)



const
    maxticks = 1000;                    (* biggest time value *)
    maxrpm = 8000;                      (* largest possible RPM *)
    ms = 2;                             (* unit of time is 500 us *)
    cylinders = 8;                      (* size of engine *)
    maxsparkretard = 30;                (* max retard angle *)
    mustrecalc = 10;                    (* 10 rpm change forces recalc *)
    stalllim = 200*ms;                  (* after 0.2 secs, stop fuel *)
    interval = 2000;                    (* interval of spark table *)
    tablemax = 15;                      (* max entry in table *)
                                        (* retard at 1000 rpm int*)
type
    rpm = 0 .. maxrpm;                  (* revolutions per minute *)
    angle = 0..45;                      (* for shaft angles *)
    ticks = 0..maxticks;                (* for time measurement *)
    delay = 0..stalllim;                (* spark delay type *)
    tableindex = 0..5;                  (* index to retard table *)
    tableentry = 0..tablemax;           (* entry in table *)
    tabletype = array [tableindex] of tableentry; (* a retard table *)
                                        (* spark retard table *)
                                (* i.e at 2000 rpm, 12 degree retard*)
value sparktable = tabletype(15,12,8,6,2,0);
```

```
(*
        Rule function used in proofs concerning spark retard table
*)
rule function nonincreasing(a: tabletype; i,j: tableindex): boolean;
        begin end;


(*
        Monitor for interlocking  -  within the monitor processing is
        sequential.

        There is no process associated with this monitor; it exists only
        to protect the shared variables.  The processes clockprocess and
        shaftprocess use the procedures exported from this monitor.
*)
monitor excluder priority 2;
exports
    doclocktick,                        (* called from clock monitor *)
    doshaftpulse;                       (* called from crankshaft monitor
imports
    nonincreasing,
    rpm, angle, ticks,
    tableindex, tableentry, tabletype, tablemax,
    delay,
    sparktable,
    mustrecalc,
    maxticks,
    maxrpm,
    interval,
    stalllim,
    excluder,
    ms, cylinders, maxsparkretard;
```

```
(*
        hardware interfaces
*)
type engineinterface = device
        fuelpumpswitch: boolean;        (* fuel pump on-off *)
        firespark: boolean;             (* store into here to fire *)
        end;



                                        (* max spark delay *)
const minrps = 1;                       (* minimum revs/second *)
                                        (* largest time per rev *)
      maxtimeperrev = (ms * 1000) div minrps;
                                        (* worst-case spark delay *)
      maxsparkdelay = stalllim;         (* worst-case spark delay *)


var
(*
        monitor global variables
*)
    engine: engineinterface[01000];     (* engine hardware i/o *)

    sparkdelay: 0..maxsparkdelay;       (* between pulse and spark*)
                                        (* angle: pulse to spark *)
    enginespeed: rpm;                   (* actual engine speed *)
    fuelpumpon: boolean;                (* last orders to fuel pump *)
    ticksuntilspark: 0..maxsparkdelay;  (* ticks until next spark needed
    tickssinceshaft: ticks;             (* ticks since crankshaft pulse *
    oldenginespeed: rpm;                (* speed at last spark recalc *)
(*
        global proof variables

        these have no existence in the operational program
        and can be used only in verification statements.
*)
    cylssincespark: extra integer;
    tickssincespark: extra integer;
```

```
invariant
(*
        The follogin invariants are invariants of the excluder
        module.  These invariants must be true whenever control
        is not in the excluder module.
*)


(*
        The following invariants describe real-world constraints to
        be proved about the program.
*)
                        (* if fuel pump is on, spark must occur soon*)
    fuelpumpon implies (tickssincespark < (1000*ms));

                        (* fuel pump must be disabled if the
                           engine is not rotating *)
    (enginespeed < rpm(1)) implies (not fuelpumpon);

                        (* a spark must be issued for each cylinder pulse
    cylssincespark <= 1;
```

```
(*
        The following invariants are needed to help the proof process.
        They must be proven; they are not accepted as given.
*)
                                (* either we have a spark scheduled or
                                   we haven't seen a cylinder pulse
                                   since the last spark
                                *)
           ((cylssincespark > 0)
            and (ticksuntilspark > 0))
         or ((cylssincespark = 0)
             and (ticksuntilspark = 0));
                                (* the invariants below were introduced
                                   during the task of making the program
                                   verifiable *)
                                (* if engine is running,
                                   spark delay must be set *)
    (enginespeed > 0) implies (sparkdelay > 0);
                                (* also true for last time around *)
    (oldenginespeed > 0) implies (sparkdelay > 0);
                                (* consistency of timers *)
    (cylssincespark = 0) implies (tickssinceshaft >= tickssincespark);
                                (* upper bound on tickssincespark *)
    (enginespeed > 0) implies
        ((tickssinceshaft + 2*stalllim) >= tickssincespark);
                                (* consistency of tickssincespark *)
    (enginespeed > 0) implies
        ((tickssincespark + ticksuntilspark) <= 2*stalllim);
                                (* if timeout, stalled *)
    (tickssinceshaft > stalllim) implies (enginespeed = 0);
                                (* fuel pump locked to engine speed *)
    (enginespeed > 0) = fuelpumpon;
                                (* old speed reset after stall *)
    (oldenginespeed = 0)  =  (enginespeed = 0);
                                (* definedness conditions *)
    defined(enginespeed);
    defined(ticksuntilspark);
    defined(tickssinceshaft);
    defined(fuelpumpon);
    defined(cylssincespark);
    defined(tickssincespark);
    defined(oldenginespeed);
    defined(sparkdelay);
    defined(excluder);
```

```
(*
        spark  --  fire spark and update counters
*)
procedure spark;
    exit tickssincespark = 0;
         cylssincespark = 0;
begin
    engine.firespark := true;               (* fire spark *)
    proof tickssincespark := 0;             (* update proof variables *)
    proof cylssincespark := 0;
end (* spark *);


(*
        fuelpumpset  --  check engine speed and set fuel pump
*)
procedure fuelpumpset;
exit fuelpumpon = (enginespeed.old > 0);     (* on iff engine running *)
begin
    fuelpumpon := enginespeed > 0;   (* turn on iff engine running *)
    engine.fuelpumpswitch := fuelpumpon;(* DEVICE I/O *)
end (* fuelpumpset *);
```

```
(*
    doclocktick  --  called from clock monitor on every tick

    this procedure issues the spark command when required, and
    turns the fuel pump on and off based on engine rpm.
*)
procedure doclocktick;
begin
     proof if tickssincespark < maxticks then (* count time for
                                               spark proof *)
        tickssincespark := tickssincespark + 1;

     if tickssinceshaft < maxticks then     (* avoid timer overflow *)
         tickssinceshaft :=       (* used to compute inverse of rpm *)
         tickssinceshaft + 1;

     if tickssinceshaft >= stalllim then (* check for stalled engine *)
     begin
         enginespeed := 0;                  (* engine is not rotating *)
         oldenginespeed := 0;               (* forget past history *)
          proof cylssincespark := 0;        (* forget about spark history
         proof tickssincespark := 0;        (* forget about spark history
         tickssinceshaft := 0;              (* forget about crank history
         ticksuntilspark := 0;              (* unschedule spark *)
         end;

                                            (* spark timing *)
     if ticksuntilspark > 0 then            (* if spark scheduled *)
     begin                           (* count down time until spark *)
        ticksuntilspark := ticksuntilspark - 1;
        if ticksuntilspark = 0 then spark;  (* fire spark if time *)
     end;
     fuelpumpset;                           (* decide fuel pump on/off *)
end; (* doclocktick *)
```

```
(*
        recalcretard  --  recalculate the spark offset

    This is called only when engine RPM changes by a significant amount.
    The calculation is by linear interpolation from a table.
*)
procedure recalcretard;
exit (enginespeed.old > 0) implies (sparkdelay > 0);
     oldenginespeed = enginespeed.old;
                                          (* retard at 1000 rpm int*)
type tdiff = 0..tablemax;                 (* table difference *)
var
    low, high: tableentry;                (* value in table *)
    diff: tdiff;                        (* difference between neighbors
    offset: 0..interval;                  (* offset from start of entry
    i: tableindex;                        (* which entry *)

const tickspersec = 1000*ms;              (* ticks per second *)
                                       (* really need fixed-point here *)
      k = (tickspersec div 360) * 60;  (* convert degrees to rpm-ticks *)
var sparkretard: angle;                (* calculated spark retardation *)
    delaywork: 0..45*k;                   (* largest possible value *)
begin
                                        (* force case analysis for table
    assert(nonincreasing(sparktable,0,0));  (* table is monotonic *)
    assert(nonincreasing(sparktable,0,1));  (* table is monotonic *)
    assert(nonincreasing(sparktable,0,2));  (* table is monotonic *)
    assert(nonincreasing(sparktable,0,3));  (* table is monotonic *)
    assert(nonincreasing(sparktable,0,4));  (* table is monotonic *)
    assert(nonincreasing(sparktable,0,5));  (* table is monotonic *)

    i := tableindex(enginespeed div interval); (* calc table index *)
    assert(sparktable[i] >= sparktable[i+1]); (* goal for rule *)
    offset := enginespeed mod interval;  (* offset from last entry *)
    low := sparktable[i];                 (* table entry from low side *)
    high := sparktable[i+1];              (* table entry from high side *)
    assert(high <= low);                  (* this is a decreasing table *)
    diff := tdiff(low - high);            (* difference in this interval *
                                          (* linear interpolation *)
    sparkretard := angle(high + (diff * offset) div interval);
    assert(sparkretard <= maxsparkretard);  (* not too much *)
                                          (* compute delay until spark
    sparkdelay := 0;                      (* assume 0 (no spark) *)
    if enginespeed > 0 then begin         (* if engine turning *)
                                          (* compute spark delay *)
        delaywork := ((k*sparkretard) div enginespeed) + 1;
```

```
                                          (* avoid oversize delay
                                             at low rpm *)
        if delaywork <= (stalllim div 2) then sparkdelay := delay(delaywo
                                         else sparkdelay := delay(stallli
    end;
    oldenginespeed := enginespeed;          (* save speed at last calc *)
end; (* recalcretard *)
```

```
(*
        doshaftpulse  --  handle crankshaft pulse
*)
procedure doshaftpulse;
var speedchange: integer;                   (* local for calculation *)
    rpmwork: 0..20000;                      (* working RPM *)
begin
    if ticksuntilspark > 0 then begin       (* if spark still in future *
                                    (* TROUBLE: clock may have failed *
        spark;                          (* force spark now, poorly timed *
        end;

    assert(cylssincespark = 0);             (* must not miss spark *)
    proof cylssincespark        (* we try to prove this never reaches 2 *
           := cylssincespark + 1;
                                            (* engine speed computation *
    if (tickssinceshaft > 0) then begin
                                            (* compute new rpm *)
        rpmwork := 1 + (60*ms*(1000 div cylinders)) div
                tickssinceshaft;
        assert(rpmwork > 0);            (* must be running if cyl pulse *
        if rpmwork > maxrpm then        (* limit measured engine speed *)
            enginespeed := maxrpm
        else enginespeed := rpm(rpmwork);
        tickssinceshaft := 0;               (* clear shaft timer *)
    end else begin                      (* TROUBLE: probable clock fail *)
        enginespeed := rpm(1);              (* assume minimum RPM *)
        end;
                                            (* recalc spark if speed chg*
    if oldenginespeed = 0 then begin        (* engine just started *)
        ticksuntilspark := 0;           (* clear all timers and counters *
        tickssinceshaft := 0;
        proof tickssincespark := 0;
        recalcretard;                       (* recalc spark retardation *
    end else begin                          (* engine did not just start
        speedchange := enginespeed - oldenginespeed; (* calc speed change
                                            (* take abs value *)
        if speedchange < 0 then speedchange := - speedchange;
        if (speedchange > mustrecalc) then  (* if big change *)
            recalcretard;                   (* go recalculate spark *)
        end;                                (* end time to recalculate *)

    ticksuntilspark := sparkdelay;          (* schedule next spark *)
    fuelpumpset;                            (* turn fuel pump on *)
end (* doshaftpulse *);
```

```
begin
(*
        initialization
*)
    enginespeed := 0;                       (* start with engine stopped
    ticksuntilspark := 0;
    tickssinceshaft := 0;
    sparkdelay := 0;
    fuelpumpon := false;
    proof cylssincespark := 0;
    proof tickssincespark := 0;
    oldenginespeed := 0;                    (* original speed is zero *)
end; (* excluder *)


(*
        shaft signal process - once per cylinder time
*)
monitor shaftprocess priority 2;
imports excluder, rpm, doshaftpulse;
entry defined(excluder);                    (* must be defined at INIT *)
invariant defined(excluder);                (* must stay defined *)

var
    shaftpulse: signal[0002B];
                                            (* crankshaft pulse interrupt
(*
        shaft processing loop
*)
begin
    while true do begin
        wait(shaftpulse);                   (* wait for shaft pulse *)
        doshaftpulse;                       (* handle shaft pulse *)
        state(defined(excluder));           (* loop invariant *)
    end;                                    (* end forever loop *)
end; (* shaftprocess *)
```

```
(*
        clock monitor

        all the processing is done in the shaft monitor.
*)
monitor clockprocess priority 2;
imports excluder, doclocktick;              (* from shaft *)
entry defined(excluder);                    (* must be defined at INIT *)
invariant defined(excluder);                (* must stay defined *)
var hardwareclock: signal[0004B];           (* clock interrupt *)
begin
    while true do
        begin wait(hardwareclock);          (* wait for clock interrupt *
              doclocktick;                  (* handle clock interrupt *)
              state(defined(excluder));     (* loop invariant *)
        end;
end;                                        (* end of clock monitor *)


(*
        main program
*)
begin
    init excluder;                          (* initialize variables *)
    init shaftprocess;                      (* start crankshaft process *
    init clockprocess;                      (* start clock process *)
end.
```

Verification of this program takes about one hour and seven minutes on a VAX 11/780, without any previous history being available. This does not include the building of some necessary rules about the function *nonincreasing* with the Rule Builder. are sufficient to verify this program.

```
Verifying clockprocess
No errors detected

Verifying excluder
No errors detected

Verifying excluder-doclocktick
No errors detected

Verifying excluder-doshaftpulse
No errors detected

Verifying excluder-fuelpumpset
No errors detected

Verifying excluder-recalcretard
No errors detected

Verifying excluder-spark
No errors detected

Verifying shaftprocess
No errors detected

Verifying simpleengine
No errors detected
```

6. Rules

It is possible to prove quite complex things about programs with the Verifier. In order to accomplish this, the user must define *rule functions* which represent the properties to be proven and must prove rules about them using the *rule builder.* We will illustrate this with a simple example, going into considerable detail on how to go about doing such things.

6.1 An introduction to rules

Consider the following simple program fragment.

```
type tabix = 1..100;
type tab = array [tabix] of integer;
var table1: tab;
    i,j: tabix;
...
for i := 1 to 100 do begin
    table1[i] := 0;
    end;
...
assert(table1[j] = 0);  { table1[j] must be 0 }
```

Here we have cleared an array to zero, and somewhere further along in the program we need to be sure that a specific element in the array is zero. To do this we will need some way to express the concept that all the elements in the array are zero. Since there is no built-in way in Pascal to talk about the values of all the elements in an array in a collective sense, it is necessary to introduce a means for doing this.

In the is case we introduce a *rule function* called *allzero.* We define it in Pascal-F with the declaration

```
rule function allzero(a: tab; i,j: tabix): boolean; begin end;
```

Here, the keyword **rule** identifies this as a rule function. Rule functions are used only for verification purposes and are ignored by the compiler. Rule functions may not have function bodies or entry and exit assertions; they have only a minimal declaration as shown in the example.

Rule functions are useful only when some rules about them have been proven. We will define *allzero* informally as being true if the array elements of *a* are all zero from element *i* to element *j* inclusive and false otherwise. Some useful rules for *allzero* are

(j < i) implies allzero(a,i,j)

which says that *allzero* is vacuously true if the the upper bound is less than the lower bound;

(i = j) implies (allzero(a,i,j) = (a[i] = 0))

which tells us that if both bounds are the same, the value of allzero is equal to true if the value at the bounds is true and false otherwise;

(allzero(a,i,j) and (a[j + 1] = 0)) implies allzero (a,i,j + 1)

which states that if *allzero* is true from *i* to *j* and *a[j + 1]* is equal to zero then *allzero* is true from *i* to *j + 1;*

(allzero(a,i,j) and ((x < i) or (x > j)))
implies allzero (<a,x,v>,i,j),

(where the strange notation <*a,x,v*> means ‘‘the array *a* with element *x* replaced by *v* which says that if *allzero* was true from *i* to *j,* storing into an element *x* outside the range *i*

to *j* will not destroy the *allzero* property; and finally

((allzero(a,i,j) and (x >= i) and (y <= j)) implies (a[i] = 0)

which allows us to use the information that allzero is true for a range of values to prove that a specific value is zero.

For the time being, we will ignore where these rules came from and will concentrate on what can be done with them. Going back to our program, it is clear that we are going to want to prove

allzero(table1,1,100)

at the end of the **for** loop. This implies that a loop invariant, a **state** statement, will be required to describe the situation which is true at each iteration of the loop.

One minor complication is that it is not obvious to the Verifier that storing into *table1[i]* does not change the fact that *allzero* is true from *1* to *i-1*. We are going to need a rule that tells the Verifier that out-of-bounds stores don't cause *allzero* to become false. Also, the rule handler in the Verifier only uses one rule between any two statements. We thus will have to put an assertion in the program that the previous STATE is still true after the assignment statement. Since *i* has increased by 1, this assertion will be that *allzero* is true from *1* to *i-1*. So now let us see the program with the addition of the required assertions.

```
type tabix = 1..100;
type tab = array [tabix] of integer;
rule function allzero(a: tab; i,j: tabix): boolean; begin end;
var table1: tab;
    i,j: tabix;
...
for i := 1 to 100 do begin
    table1[i] := 0;
    assert(allzero(table1,1,i-1));
    state(allzero(table1,1,i));
    end;
assert(allzero(a,1,100));
...
assert(table1[j] = 0);  { table1[j] must be 0 }
```

Each time through the loop, *allzero* becomes true for one more element, and *allzero* is then true from 1 up to the loop index.

Given the rules described, the verifier is able to prove all the assertions in the program fragment automatically.

Typically, the user will be provided with a knowledge base containing a library of rules which cover most common situations in programming, and will be able to proceed much as shown above. In the next chapter, we will return to the example above and show in

detail how the rules for it are proven.

6.2 Rules and the Verifier

6.2.1 How the Verifier applies rules

Rules are almost always of the form

A implies B

where *A* is referred to as the *hypothesis* and *B* as the conclusion. The Verifier, when trying to prove something such as

```
assert(allzero(a, i, j));
```

where *allzero* is a rule function, will search the database for rules with conclusions of the form

allzero(x,y,z)

and will then apply the rule by *binding* each free variable in the rule to the corresponding expression in the form being proven. This process is called *instantiating* the rule, because the general form of the rule has now been applied to a specific instance. The Verifier will try every applicable rule in every legitimate way, but will not apply a rule to a form introduced by a rule. In other words, the Verifier applies rules only one deep. It is the user's job to add assertions to the source program so that between any two assertions the application of only one rule is required. How to do this was described in detail in chapter 2.

6.2.2 How the rules get to the Verifier

Rules are created with the Rule Builder (described in the next chapter) and made available to the Verifier with the **putrules** utility program. The Rule Builder creates a **knowledge base** file, which may be used for verifying several different programs. The rules being used in a given verification must be placed in a **ruledatabase** file. There is one such file for each program being verified, and this file resides in the directory of work files used by the verifier for each program being verified. The directory has the name

```
<programname>_d
```

where the program being verified has the name

```
<programname>.pf
```

This directory is created by the Verifier the first time an attempt is made to verify a given program, and is never deleted by the Verifier. The **ruledatabase** and **history** files, along with all the scratch files used during verification, reside in this directory, which is managed entirely by the Verifier. The user should not alter any file in the work file directory at any time.

6.2.2.1 The putrules utility

The **putrules** utility reads a knowledge base file and creates a **ruledatabase** file. It is invoked with the call

```
putrules <knowledge base> <program>_d
```

where **program** is the name of the program (not including the trailing **.pf**) being verified. Putrules can only be run after the verification of the program has been tried at least once and the Verifier's work directory **<program>_d** created by the Verifier. After any change to the knowledge base, it is necessary to rerun **putrules** to make the changes in the knowledge base available to the Verifier.

If the changes to the knowledge base included the alteration or deletion of any rule function definition, **putrules** will print a message so stating and will clear the history of successfully verified routines, so that a full reverification will be performed the next time the Verifier is invoked. This check is required to insure soundness.

6.2.3 Standard knowledge bases

Standard knowledge bases may be created. A knowledge base may contain information useful for verifying more than one program. For example, the information about **allzero** above would be useful in any program that cleared arrays to zero. One useful knowledge base is the Rule Builder's built-in knowledge base. This knowledge base is called **verifier** and is usually sufficient (and necessary) for programs which contain no user-provided rule functions but do contain instances of the 3-argument form of **DEFINED** for proving the definedness of parts of arrays.

Usually a copy of this knowledge base is maintained in a well-known place on each system on which the Verifier is installed. But a copy of this knowledge base can be obtained, if required, by invoking the Rule Builder and, without proving any new theorems, using the Rule Builder command

```
(MAKE-LIB 'verifier)
```

which will create the knowledge base files *verifier.lib* and *verifier.lispin the current directory.*

7. The rule builder

In the previous chapter, the concept of verification using *rules* was introduced. These rules must be created by the user and proven with machine assistance. The rule builder is a tool used with the Verifier proper to construct sound new rules and thus provide the Verifier with more knowledge.

7.1 Introduction

Using the Rule Builder requires a substantially greater level of expertise in both formal mathematics and verification than does using the Verifier proper.

The Rule Builder is a version of the Boyer-Moore theorem prover initialized with a knowledge base compatible with the Verifier's built-in knowledge. It is a difficult

program to use, and requires some training in formal logic to use successfully. Not all users of the Verifier need be familiar with the Rule Builder, but users must have access to someone with this expertise for assistance when a verification requires a new rule.

7.1.1 Required reading

Before attempting to use the rule builder, it is necessary to become familiar with three documents. The first of these is the book *A Computational Logic*, by Robert S. Boyer and J. Strother Moore, published in 1979 by Academic Press, New York, NY. This book describes the theory upon which the prover is based. Chapters 1, 2, and 3 should be studied closely, and reading the entire text while trying out some of the examples is advisable.

Some knowledge of the Franz Lisp implementation of Lisp is required. The reference manual is provided with the Berkeley UNIX distribution, and is also available from Franz, Inc. of Berkeley, California. However, anyone with a working knowledge of Lisp will probably not require this manual.

Finally, the manual *A Theorem Prover for Recursive Functions: A User's Manual* published in 1979 as Report CSL-91 by SRI International of Menlo Park, CA., is useful. This manual describes the mechanics of running the theorem prover. The key parts have been extracted and appear in the next few pages, but the theorem prover user's manual goes into more detail and should be on hand.

The prover has changed somewhat since the 1979 manual. We will try to cover the important differences in this manual.

7.2 An introduction to the Rule Builder

In *A Computational Logic* [BOYER79], Boyer and Moore describe a formal logic based on recursive functions, and they present a large number of techniques for discovering proofs in that theory. These techniques are implemented in their theorem prover. This chapter of the verifier manual is an adapted version of Boyer and Moore's theorem prover manual.

7.2.1 Teaching the theorem prover

While using the rule builder the user will spend most of his time teaching the system about the concepts he defines and their relationships to other defined concepts. The system is taught by defining functions and suggesting lemmas for it to prove and remember for future use.

The system uses axioms and previously proved lemmas in four distinct ways. The system does not decide automatically how to use a given theorem; whenever any new theorem is introduced, the user must specify how the lemma is to be used by providing the system with a list names drawn from the following keywords.*

---

* The *induction* lemma type has been discontinued. It is now necessary to use manually-provided *hints* to help the prover with difficult inductions.

rewrite	lemmas are used to rewrite terms. Most lemmas are rewrite lemmas.
elim	lemmas are used to replace certain complex terms by single variables.
generalize	lemmas are used to guide the theorem prover when it looks for stronger induction hypotheses.

Any lemma with an empty list of lemma types will never be used again by by the system.

The theorem prover is very sensitive to the syntactic form chosen by the user to represent each new fact. For example, a rewrite lemma of the form

(implies (and p q) (equal r s))

is used to rewrite instances of *r* to *s* provided that the system can first establish *p* and then *q.* This is the most common form of lemma. Note the asymmetry between hypothesis and conclusion, and between left and right hand sides of the conclusion. In fact, because the system must limit the resources it is willing to spend establishing *p* and *q,* even the order of the hypotheses is relevant to the system. Thus, the above rewrite lemma causes different behavior than any of the following logically equivalent formulae:

(implies (and p q) (equal s r))

(implies (and p (not (equal r s))) (not q))

(implies (and q p) (equal r s))

To become an effective user of the system one must understand how the commands influence the behavior of the system. It is possible to infer the meaning of the various lemma types after enough hands-on experience with the system. (Boyer and Moore add the comment here ‘‘It is also possible to infer the structure of a brick wall by battering it down with your head’’.)

7.2.1.1 Events, Dependencies, and Commands

The insertion of a definition or lemma into the knowledge base is called an **event.** All events have names. Some events, such as definitions of new functions, are naturally associated with a name (e.g., the name of the function defined); others, such as theorems, are given names by the user. See the section on syntax below.

The basic theorem-prover commands are those that create new events: the definition of a new function, and the proof and storage of a new theorem. The commands that create new events are **dcl, defn, prove-lemma,** and **move-lemma.**

Events are related to each other by logical dependencies. For example, the admission of a certain formula as a theorem depends on all of the functions and lemmas used in the

proof of the theorem. Similarly, the admission of a new recursive function definition depends not only upon all of the previously introduced concepts used in the definition, but also upon the functions and lemmas used to prove that the proposed ‘‘definition’’ truly defines a function.

Thus, the theorem-prover’s knowledge base is actually a noncircular, directed graph of events. The theorem prover’s performance is largely determined by its knowledge base. The Rule Builder is initialized with a knowledge base with definitions and lemmas which define the basic objects of Pascal-F verifications, integers and arrays, and provide a reasonable set of knowledge about arithmetic and array operations. The user will need to add his own definitions and prove theorems about them. It is possible to dump the system’s knowledge base to a ‘‘library file’’ to save the system’s state from one session to the next, and to provide information to the Verifier proper.

After proving several theorems, the user finds that one of his earliest defined concepts was inconveniently or inappropriately defined, the user can undo that definition (using the **undo-name** or **undo-back-through** commands) and lose only those results whose meaning or logical validity may depend on that definition.

7.2.2 Error handling

If one tries to execute an inappropriate command (e.g., assign the same name to two different events, or attempt to define a function in terms of unknown concepts) self-explanatory error messages will be printed. The system checks for over 100 errors and has an error handling mechanism designed to keep the theorem proving machine in a consistent state. For example, when an new command is processed, all possible errors are checked before the first change is made to the data base, since an abortion midway through the update would leave the machine in an unacceptable state.

Error messages are grouped into three classes; WARNING, ERROR, and FATAL ERROR messages. Warnings arise when the system has detected something unusual but not logically incorrect. For example, the system prints a WARNING message if the user defines a function but do not refer to one of the formal parameters in the body of the function. After printing a WARNING message, the system continues normal execution.

ERROR messages result from true errors in the sense that the system cannot continue until the error is repaired, but the error can be repaired by editing a formula or changing a name. When such an error occurs the system prints an explanatory error message and then returns to the LISP command prompt level, discarding the failed command.

FATAL ERROR messages occur when system resources are exhausted or when internal checks indicate the presence of inconsistency in the data base or bugs in the theorem prover itself. It is usually not possible to proceed past a fatal error. When a FATAL ERROR is observed, it should be reported as described in the section on reporting verifier problems.

Despite the precautions taken in the theorem prover, it is possible to get the system in an illegal state by aborting a command while the data base is in the process of being changed. Ctl-C will abort any command, but this is unsafe. However, the prover may spend hours exploring dead ends when trying some proofs. It is thus necessary to abort

commands on some occasions. The following cautions apply:

- **dcl, make-lib, move-lemma, note-lib,** and **undo-name** should never be aborted while running.
- **dependent-events, events-since prove, ppe,** and **chronology** may be aborted at any time without harm.
- **defn** and **prove-lemma** can be aborted but there is a small risk of corrupting the theorem prover's database, if the **defn** or **prove-lemma** had in fact succeeded and the database was in the process of being updated.

Boyer and Moore recommend that the result of any proof where a command was aborted be considered suspect.

7.2.3 Output

The theorem prover prints an English description of what it is doing as it proceeds. The sample session, shown below, shows what this is like. Normally, the output goes to the user's terminal, but can be diverted.

7.2.4 Syntax

All formulas in Boyer-Moore logic are written in a LISP-like prefix notation. This notation is fully described in chapter III of *A Computational Logic.*

7.2.4.1 Functions

The functions usable in this notation are those defined in the section ''The Built-In Knowledge Base'' and any new user-defined functions the user introduces with the **defn** and **dcl** commands. Functions have a fixed number of arguments as specified in the definition of the function.*

7.2.4.2 Variables

The variables used in this notation are free variable names and have no relationship to names used in Pascal-F programs. Names of variables, and of new functions, may be composed of the characters A-Z, a-z, 0-9, ''.'', and ''!''. Upper case letters are converted to lower case letters.

7.2.4.3 Constants

The available constants are the integers, written in the usual way, and the explicit constants shown below.

(true)	Boolean true.
t	Alternate for (true).
(false)	Boolean false.
f	Alternate for (false).

---

* Some functions are by design N-ary, for example, and and or.

(undefined)	A specific object which is not an array, an integer, or a Boolean value. It is used in building up the theory of arrays.
(empty.array)	An array-valued object, all elements of which are equal to (undefined).

Only **t, f, and the integers are typically used by the user of the Rule Builder.**

7.3 The mechanics of using the Rule Builder

7.3.1 Starting up the program

The instructions in this section apply to the UNIX version of the program.

The command

```
rulebuilder
```

invokes the Rule Builder. This is a version of the Boyer-Moore prover pre-initialized with a knowledge base compatible with the Verifier. It takes about a minute to load the program (which requires about 2 megabytes of memory.) The messages

```
Pascal-F Rule Builder of  2-JAN-86 16:02:04
[load /usr/lib/verifier.lisp]
Standard Pascal-F knowledge base loaded.
->
```

indicate that the system is ready for commands.

If a previous session with the Rule Builder has been used to produce a knowledge base, that knowledge base can be used as a starting point for a new Rule Builder session by invoking the Rule Builder with the command

```
rulebuilder knowledgebase
```

where *knowledgebase* is the name of a knowledge base created with the *make-lib* command.

Commands are aborted with control-C, which is sometimes unsafe, as mentioned above.

7.3.2 Commands Summary

The commands listed below are a subset of the full command list of the Boyer-Moore theorem prover. The commands listed are normally sufficient for building rules for the Verifier.

7.3.2.1 chronology

The word **chronology** , without parentheses, will display a list of the names in the current knowledge base, in reverse chronological order. Thus, the last name listed will be the name of the oldest event, which is usually **ground-zero.** The **chronology** list covers events brought in through **note-lib, so the events which make up a library may be listed.**

7.3.2.2 (dcl <name> <arglist>)

Dcl declares *name* to be an undefined function of N arguments, where N is the length of *arglist,* which must be a list of distinct, but otherwise meaningless, variable names. Functions created via *dcl* have no semantics in the Rule Builder, but may have semantics in the Verifier. The theory of uninterpreted functions does apply to functions created by dcl, which means only that if *f* is an uninterpreted function,

```
   x = y
implies
   f(x) = f(y)
```

7.3.2.3 (defn <name> <arglist> <body> [<hints>])

Defn defines a function named *name,* with formal argument list *arglist* and body *body.* The optional parameter *hints* allows the user to assist the theorem prover in validating the definition. The *arglist* must be a list of distinct variable names, and *body* must be an expression in the theory. This expression must be constructed as described under the section on syntax, and may use as variable names only the names present in *arglist.* Only previously defined functions, and the the function being defined in the defn, may be used in the *body.*

For recursive definitions, the system insists that the recursion terminate and will not accept the definition fully unless it can prove this. The system will try to prove that some measure of *arglist* gets smaller in each recursive call to *name* within *body.*

When the system cannot prove that a recursive definition terminates, a WARNING message appears stating that the definition is not well-founded. The definition is not, however, rejected by the system. It should be. The user should immediately remove it with an **undo-name** command.

The *hint* parameter to *defn* is seldom required. If the function being defined is nonrecursive, it is never required. If the function is recursive but recurses in such a way that at least one of the arguments in the recursive call is clearly less than the value at invocation, no hint should be required. For example, if the function recurses by subtracting 1 from an argument until the value becomes zero, the theorem prover will be able to satisfy itself of the soundness of the definition without difficulty. But if a function recurses by *adding* one to an argument until a limit is reached, a hint will be necessary.

Hints for *defn* are very similar to the Pascal-F **MEASURE** statement; the user must supply an expression whose value becomes smaller with each recursion. Hints have the

form of a list of **(<comparing-operator> <recursion measure>)** terms. In the usual case, where only one hint is required, the *hint* parameter has the form **((<comparing-operator> <recursion measure>)).** Suitable *comparing-operators* are **lessp,** and **lex2*.** The *recursion measure* must be some expression which, when evaluated both at entry to the recursive function and at entry to the recursive function one level deeper in the recursion, becomes smaller with each recursion. Here, ‘‘smaller’’ is defined relative to the comparing-operator chosen.

Note that **defn** definitions are actually small recursive programs. It is possible to run these programs on test data; see the **r** command.

7.3.2.4 (dependent-events <name>)

Dependent-events takes an event name (i.e. a function or lemma name) and returns the events which depend on it. If *name* is deleted with *undo-name,* all the dependents of *name* will be deleted.

Dependency is simply defined. If a **defn** *g* mentions a function *f,* then *g* depends on *f.* If the proof of lemma *x* used lemma *y* and **defn** *f,* then *x* depends on *f* and *y.*

7.3.2.5 (events-since <eventname>)

This returns a list of all the events stored since the named event. The list is based strictly on time, not dependency.

7.3.2.6 (exit)

Exit causes an exit from the Rule Builder. Any new knowledge added since the last **make-lib** is lost.

7.3.2.7 (lemmas <functions>)

This is a cross-referencing tool. A list of all lemmas which mention any function in *functions* is returned.

7.3.2.8 (make-lib <file>)

Make-lib makes a file named *file.lib* and a file named *file.lisp* which together contain the entire current knowledge base. Invoking the Rule Builder with

```
rulebuilder <file>
```

will restore the state of the Rule Builder to that in effect when the **make-lib** was executed.

Library files are of moderately large size, about 100 kilobytes, and contain not only the events but substantial amounts of internal information.

7.3.2.9 (move-lemma <name> <lemmatypes> <oldname>)

The *move-lemma* command allows the user to change the ways in which a lemma can be applied within the Rule Builder. It is used primarily to ‘‘hide’’ lemmas or definitions

---

* See *A Computational Logic* for an explaination of *lex2.*

which although correct cause the Rule Builder to pursue dead ends. Usually, *lemmatypes* is NIL, which causes the **defn** or **prove-lemma** event *oldname* to be hidden. When a **prove-lemma** event is hidden, the lemma will not be used by the Rule Builder, and when a defn event is hidden, that definition will not be opened up. Hiding a **defn** does not prevent its evaluation with the r command.

7.3.2.10 (note-lib <file>.lib <file>.lisp)

Note-lib reads in *file,* which must have been produced by **make-lib,** and reinitializes the Rule Builder with the knowledge base in that file. This is a reinitialization, not an addition; the state of the theorem prover is cleared before the read.

7.3.2.11 (ppe <eventname>)

The *ppe* command prints the event *eventname* in a tidy format.

7.3.2.12 (prove <thm>)

Prove attempts to prove the conjecture *thm,* using all the theorem-proving techniques at the system's disposal. prove prints an English-language description of the proof attempt in real-time, so the user can monitor the progress of the attempt. The result of the proof is not saved.

7.3.2.13 (prove-lemma <eventname> <lemmatypes> <thm> [<hints>])

This is the most important command in the Rule Builder. *Prove-lemma* attempts to prove a lemma as with **prove,** and if the attempt is successful, the lemma will be saved, available for use in the ways specified in the list *lemmatypes.* The allowed lemma types are rewrite, elim, and generalize. **prove-lemma** first checks to see if the syntactic form of *thm* is acceptable for the *lemmatypes* indicated. If no error is diagnosed in this pre-processing, a proof is attempted as with the prove command. If the proof succeeds, the lemma is stored under the name *eventname* and is usable with the lemma types *lemmatypes.*

If *eventname* ends in ‘‘-rule’’ or ‘‘-RULE’’, and the *lemmatypes* list includes ‘‘rewrite’’ then the event is considered a Verifier rule and, if placed in a Rule Builder library with **make-lib** and then copied to a Verifier database file with the ‘‘putrules’’ utility, will be used by the Verifier.

The optional parameter *hints* allows the user to order the theorem prover to try a specific lemma or induction at the beginning of the proof, thus affording some minimal control over the proof process. Hints are not normally necessary; it is better when possible to help the theorem prover along by proving lemmas which will then be used in later proofs. However, hints can be provided if necessary. This is an advanced feature of the prover and is not recommended for new users.

The *hints* arguement to the **prove-lemma** command, if not ommitted, must be a list of *hint entries.* There are five kinds of hints:

use	Indicates that a specific lemma is to be applied in a specific way at the beginning of the proof. The form of a **use** hint is

```
(use (event1 (v1 t1) ... (vn tn))
     ...
     (eventk (vk tk) ... (vm tm)))
```

where each *eventi* is the name of an **add-axiom, prove-lemma,** or **defn** event, each *vi* is a variable name from the definition of the event, and each *ti* is a term in the formula being proven. The user is thus explicitly requesting the application of a rule, and the user must give the exact bindings of the variables in the rule to the terms in the formula being proven.

A **use** hint is a request, not a demand; if the lemma indicated in the hint cannot be applied to the formula being proven, the hint will be ignored and the prover will proceed without it. It is thus important when using **use** hints to watch the beginning of the proof for the application of the hint.

expand	Indicates that a **defn** should be expanded at the beginning of the proof.
disable	Prevents the use of a named lemma or expansion of a named definition in the proof.
induct	Indicates the induction strategy that the prover is to use.
time	We don't know what this is for.

The *use* and *disable* hint types are sufficient for most proofs. The prover has relatively good diagnostics for incorrectly formed hints; a badly formed hint will produce a message giving the correct form and an indication of why the prover is unhappy with the hint.

7.3.2.14 (r <term>)

The *r* command evaluates *term*, which must be an expression constructed from valid built-in functions, defn functions, and constants. For example,

```
(r (plus 2 2))
```

will return 4. More usefully, if we define a function of our own,

```
(defn FACTORIAL (N)
        (if (lessp 0 N)
                (times N (FACTORIAL (difference N 1)))
                1))
```

we can then try test cases on it.

```
(r (FACTORIAL 4))
```

will return 24.

This evaluation process is quite fast. The Rule Builder generates Lisp code for each **defn** when the **defn** is created, so that definitions can be rapidly evaluated for constant values. Recursive definitions generate recursive code. Since the Rule Builder insists that definitions be provably well-founded, infinite recursion is prevented.

The Boyer-Moore theory is a constructive theory of total functions. This means that any syntactically valid variable-free expression can be evaluated. For example, FACTORIAL can be applied to T, the Boolean value true, and a consistent value will be returned. When constructing definitions, the actions for inputs of unexpected types must be borne in mind.

It is worthwhile to try out the **r** command on some of the built-in functions to develop a feel for what they do. An interesting exercise is to produce an array-valued result. For example,

```
(r (storea! (storea! (empty.array) 3 100) 5 200))
```

will produce an array-valued object with element 3 equal to 100 and element 5 equal to 200. This gives the user some insight into how arrays are represented internally.

7.3.2.15 (undo-back-through <eventname>)

Eventname and all events performed since *eventname* will be undone. This includes events not dependent on *eventname.*

7.3.2.16 (undo-name <eventname>)

Eventname and all events dependent upon it will be undone.

7.4 The built-in knowledge base

The Rule Builder, as stated before, is a version of the Boyer-Moore theorem prover pre-initialized with a knowledge base compatible with the Verifier. This knowledge base includes definitions of arithmetic for the natural numbers and the integers, Boolean and comparison functions, and a definition of arrays. It also contains about a hundred lemmas of general utility, most of which are statements about arithmetic.

7.4.1 The built-in functions

The predefined functions are as follows.

(add1 N)	Adds one, usable on natural numbers only.
(and B1 B2 ...)	N-argument and.
(difference N M)	Natural number subtraction.
(equal X Y)	Equality; usable on any type operands.
(if P X Y)	If P then X else Y. if must not appear in rules, but may be used in **defn** definitions or non-rule lemmas.
(implies P Q)	Boolean implication.
(lessp N M)	Natural number comparison.
(not P)	Boolean negation.
(numberp N)	True if value is a natural number.
(or B1 B2 ...)	N-argument or.
(plus N M)	Natural number addition.
(quotient N M)	Natural number division.
(remainder N M)	Natural number remainder.
(sub1 N)	Subtracts one.
(times N M)	Natural number multiplcation.
zero	Equivalent to 0.
(zerop N)	True if N is equal to 0.
(addi! I J)	Integer addition.
(alltrue! r)	True if all parts of the argument are true. This is a dcl to the rule builder, and has no semantics in the rule builder. The Verifier expands *alltrue!* based on type information.
(arrayp! A)	Array type predicate, true if the argument is an array.
(arraytrue! A I J)	True if the elements from I to J of array A are *alltrue!*. When `defined(A,I,J)` is written in Pascal-F source, *(arraytrue! A I J)* applied to the definedness part of A will be generated in the verification condition. This allows inductive proofs of definedness of arrays.
(booleanp! A)	Type predicate, true if A is Boolean.

(divi! I J)	Integer division.
(gei! I J)	Integer >=.
(gti! I J)	Integer >.
(integerp! I)	Type predicate, true if I is an integer.
(lei! I J)	Integer <=.
(lti! I J)	Integer <.
(mod! I J)	Integer remainder. Mod should be applied to positive numbers only, because the Verifier has no knowledge about what the result is for negative numbers. This reflects the Pascal-F implementation.
(muli! I J)	Integer multiply.
(negi! I)	Integer negation.
(numberp! I)	Type predicate, true if I is a natural number (nonnegative).
(selecta! A I)	Array subscripting function; equivalent to the Pascal form A[I].
(selectr! A F)	Record selector, equivalent to the Pascal form ‘‘A.F’’. The Rule Builder does not know about records because it lacks type information. This is a dcl uninterpreted function. The Verifier interprets *selectr!* based on the type information from the program.
(storea! A I V)	Array store function. The result of *storea!* is an array equal to A except that A[I] = V. This function has no Pascal infix-form equivalent, but is displayed in the Verifier’s log of verification conditions as ‘‘<A I V>’’.
(storer! A F V)	Record store function. The result of *storer!* is a record equal to A except that A.F = V. This function is displayed in verification conditions as ‘‘<A I V>’’, and looks just like the array store function in that form. The *storer!* function, like *selectr!*, is a dcl.
(subi! I J)	Integer subtraction.

The upper-case names are identical to those in *A Computational Logic* in both syntax and meaning. We have not changed these because they are built into the Boyer-Moore system. No rule function may be given the same name as one of the built-in names. The names ending in ‘‘!’’ represent the additional functions needed to handle Pascal-F verification conditions.

7.5 An example of rule building

In the chapter ‘‘Rules’’, we gave a sample program fragment which used a rule function called **allzero.** Verification of this piece of a program required some rules. As a concrete example, we make a simple program out of our program fragment.

```
 1  program example6;
 2  {
 3          Program fragment to demonstrate rule usage
 4  }
 5  type tabix = 1..100;
 6  type tab = array [tabix] of integer;
 7  rule function allzero(a: tab; i,j: tabix): boolean; begin end;
 8  var table1: tab;
 9      i,j: tabix;
10  begin
11      for i := 1 to 100 do begin
12          table1[i] := 0;
13          assert(allzero(table1,1,i-1));
14          state(allzero(table1,1,i));
15          end;
16      assert(allzero(table1,1,100));
17      j := 25;                          { some arbitrary value }
18      assert(table1[j] = 0);            { table1[j] must be 0 }
19  end.
```

We will make an attempt at verifying the program, knowing that the attempt will be unsuccessful, since the Verifier has no idea what **allzero** means.

```
% pasver example6.pf
```

Of course, we get diagnostic messages.

```
Pass 1:
Pass 2:
Pass 3:

Verifying example6
Could not prove {example6.pf:18} table1[(j - 1) + 1] = 0
(ASSERT assertion)
for path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} FOR loop exit

Could not prove {example6.pf:14} allzero(table1,1,i)
(STATE assertion)
for path:
    {example6.pf:11} Start of "example6"
    {example6.pf:15} Back to top of FOR loop

Could not prove {example6.pf:14} allzero(table1,1,i)
(STATE assertion)
for path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} Enter FOR loop

Could not prove {example6.pf:13} allzero(table1,1,i - 1)
(ASSERT assertion)
for path:
    {example6.pf:11} Start of "example6"
    {example6.pf:15} Back to top of FOR loop

Could not prove {example6.pf:13} allzero(table1,1,i - 1)
(ASSERT assertion)
for path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} Enter FOR loop

5 errors detected
```

We obviously need rules about **allzero.** In the previous chapters, we figured out what rules we needed. So let us build them.

We begin by invoking the rule builder

```
% rulebuilder
```

which responds with its signon message and a prompt.

```
Pascal-F Rule Builder of Wed Feb 26 19:58:22 1986
[load /u/jbn/ver/cpc6/verifier.lisp]
Default Pascal-F knowledge base loaded.
->
```

In this session, we will define the function **allzero,** which is a predicate for testing whether an array is composed entirely of zero elements between two subscript bounds. The definition is a recursive function; **allzero** is true vacuously if J is less than I, otherwise we recurse, checking each element, until J is less than I.

```
-> (defn allzero
      (a i j)
      (if (lessp j i)
          t
          (and (allzero a (add1 i) j)
               (equal (selecta! a i) 0))))

WARNING:  The recursion in allzero is unjustified.


Warning:  The admissibility of allzero has not been established.
We will assume that there exists a function satisfying this
definition.  An induction principle for this function has also
been assumed, corresponding to the obvious subgoal induction
for the function.  These assumptions may render the theory
inconsistent.

Note that (or (falsep (allzero a i j)) (truep (allzero a i j)))
is a theorem.

************** F  A  I  L  E  D **************


[14.183333 0.4166669999999992 ]
nil
```

This is no good. We must not accept this definition or our theory might be unsound. Although the prover has (grudgingly) stored the definition, we want to delete it and try again. So we use the **undo-back-through** command to delete the definition of **allzero.**

```
-> (undo-back-through 'allzero)
(defn allzero (a i j) (if (lessp j i) t (and (allzero a
(add1 i) j) (equal (selecta! a i) 0))))
```

Actually, there is nothing wrong with our definition of **allzero.** It is just that the theorem prover isn't smart enough to figure out that the recursion terminates. There are two ways to deal with this problem; one is to rewrite the definition so that the theorem prover can figure this out by itself, and the other is to provide a hint. The first approach could be applied by rewriting **allzero** so that it recursed by subtracting 1 from *j* on each iteration rather than adding 1 to *i.* The theorem prover has no trouble understanding that subtracting 1 repeatedly with a test for **lessp** in the right place must lead to termination.

But for purposes of illustration we're going to bull our way through with a hint. As mentioned in the Command Summary section for **defn,** a hint for a **defn** is a lot like the Pascal-F **MEASURE** statement. We need an expression which gets smaller with each recursion. The expression *(difference (add1 j) i)* will do the job. We need the **add1** because **difference** returns a natural number; a negative value is not possible. We thus must bias the value of *j* to avoid trouble for the case where *j* is one less than *i.*

Our hint will have the form

```
(lessp (difference (add1 j) i))
```

indicating that we want *(difference (add1 j) i)* used as the recursion measure and **lessp,** as usual, used as the well-founded relation.

```
-> (defn allzero
      (a i j)
      (if (lessp j i) t
          (and (allzero a (add1 i) j)
               (equal (selecta! a i) 0)))
      ((lessp (difference (add1 j) i))))

     Linear arithmetic establishes that the measure (difference
(add1 j) i) decreases according to the well-founded relation
lessp in each recursive call.  Hence, allzero is accepted under
the principle of definition.  Observe that:
      (or (falsep (allzero a i j))
          (truep (allzero a i j)))
is a theorem.

[ 3.2 0.25 ]
allzero
```

It succeeds; the definition is valid. It has now been proven that the recursive definition cannot loop infinitely. The theorem prover also notes that **allzero** is Boolean-valued, which it may find useful later.

We ask the Rule Builder to print the definition of **allzero** to illustrate the ppe command.

```
-> (ppe 'allzero)
```

and the definition is printed in suitably indented form, with the hint included.

```
(defn allzero
      (a i j)
      (if (lessp j i)
          t
          (and (allzero a (add1 i) j)
               (equal (selecta! a i) 0)))
      ((lessp (difference (add1 j) i))))
nil
```

Incidentally, we could have defined **allzero** so that it recursed downward, and the prover would still be able to prove every lemma proved in this session. In many ways, this would have been an easier approach; we would not have needed the hint in the **defn** command that defined **allzero.**

Let us now test out our new definition. We have defined a function and can now run it on

some test data. The r command is used to run **allzero** on an array in which element 2 is 0 and element 3 is 0. (Remember that **(storea! A I V)** is equal to the array **A** except that element **I** has been replaced by the value **V**) The form **(empty.array)** is simply the array of no elements.

```
-> (r (allzero (storea! (storea! (empty.array) 2 0) 3 0) 2 3))
```

The system responds with

```
t
```

which is what we want. Let us try an array which is not all zero.

```
-> (r (allzero (storea! (storea! (empty.array) 2 1) 3 1) 2 3))
```

```
f
```

Another test case; an array with one zero element; is it all zero from 2 to 2?

```
-> (r (allzero (storea! (storea! (empty.array) 2 0) 3 1) 2 2))
```

```
t
```

It is. Finally, we check out the case where the upper bound of the **allzero** is less than the lower bound.

```
-> (r (allzero (storea! (storea! (empty.array) 2 1) 3 1) 3 2))
```

```
t
```

This, also, seems to work.

With our definition in good shape, we can now try to prove some theorems about it. Our first lemma will be that if the lower bound exceeds the upper bound in the **allzero** call, then **allzero** is vacuously true.

```
-> (prove-lemma allzero-void-rule
            (rewrite)
            (implies (and (arrayp! a)
                          (numberp i)
                          (numberp j)
                          (lessp j i))
                     (allzero a i j)))
```

Note that the name of the lemma, **allzero-void-rule,** ends in **-rule** which will later make this rule available to the Verifier. The theorem prover now proceeds with the proof, which, given the definition, ought to be trivial.

```
This conjecture simplifies, opening up allzero, to:

      t.

Q.E.D.

[ 7.683333000000001 0.06666699999999916 ]
allzero-void-rule
```

It is trivial; the proof succeeds in 7.6 seconds (this is on a Sun II) and the event **allzero-void-rule** is stored.

We also need a rule to handle the case where both bounds of **allzero** are equal. This, too, should be trivial. We type in our lemma

```
-> (prove-lemma allzero-single-rule
             (rewrite)
             (implies (and (arrayp! a)
                           (numberp i)
                           (numberp j)
                           (equal i j))
                      (allzero a i j)))
```

and the theorem prover goes to work.

This formula simplifies, using linear arithmetic, rewriting with allzero-void-rule and x-not-less-than-x, and expanding allzero, to:

```
(implies (and (arrayp! a) (numberp j))
         (equal (selecta! a j) 0)),
```

which we will name *1.

We will appeal to induction. The recursive terms in the conjecture suggest two inductions. However, they merge into one likely candidate induction. We will induct according to the following scheme:

```
(and (implies (and (array-recognizer a)
                   (equal a (empty-array)))
              (p a j))
     (implies (and (array-recognizer a)
                   (not (equal a (empty-array)))
                   (or (not (numberp
                              (array-elt-subscript a)))
                       (equal (array-elt-value a)
                              (undefined))))
              (p a j))
     (implies (and (array-recognizer a)
                   (not (equal a (empty-array)))
                   (not (or (not (numberp
                                  (array-elt-subscript a)))
                            (equal (array-elt-value a)
                                   (undefined))))
                   (equal (array-prev a) (empty-array)))
              (p a j))
     (implies (and (array-recognizer a)
                   (not (equal a (empty-array)))
                   (not (or (not (numberp
                                  (array-elt-subscript a)))
                            (equal (array-elt-value a)
                                   (undefined))))
                   (not (equal (array-prev a)
                               (empty-array)))
                   (p (array-prev a) j))
              (p a j))
     (implies (not (array-recognizer a))
              (p a j))).
```

Linear arithmetic and the lemma array-prev-lessp can be used to show that the measure (count a) decreases according to

```
the well-founded relation lessp in each induction step of the
scheme.  The above induction scheme generates six new goals:

Case 6. (implies (and (array-recognizer a)
                      (equal a (empty-array))
                      (arrayp! a)
                      (numberp j))
                 (equal (selecta! a j) 0)),

  which simplifies, unfolding array-recognizer, arrayp!,
  equal, array-prev, array-elt-value, array-elt-subscript,
  and selecta!, to the formula:

        (not (numberp j)).

  Eliminate the irrelevant term.  This produces:

      f.

Need we go on?

************** F  A  I  L  E  D **************
```

The theorem prover stops, after about a minute of work, and reports failure. What went wrong? We can display the failed theorems in this session by typing

```
-> (pp failed-thms)
```

to which the theorem prover replies

```
    (setq failed-thms
          '((implies (and (arrayp! a)
                          (numberp i)
                          (numberp j)
                          (equal i j))
                     (allzero a i j))
            (defn allzero
                  (a i j)
                  (if (lessp j i)
                      t
                      (and (allzero a (add1 i) j)
                           (equal (selecta! a i) 0)))
                  nil)))
t
```

We get to see our previous failure with the definition of **allzero** as well as our latest problem. The problem is obvious; we aren't testing anything for zero in the hypotheses of the theorem, so we can't possibly expect it to prove *allzero* true in the conclusion. We are missing a hypothesis. Let us try again.

```
    -> (prove-lemma allzero-single-rule
                (rewrite)
                (implies (and (arrayp! a)
                              (numberp i)
                              (numberp j)
                              (equal i j)
                              (equal (selecta! a i) 0))
                         (allzero a i j)))
```

We have added the hypothesis *(equal! (selecta! a i) 0)* and the theorem prover is now able to prove this quite easily.

```
This conjecture simplifies, using linear arithmetic, rewriting
with allzero-void-rule and x-not-less-than-x, and unfolding the
functions equal and allzero, to:

      t.

Q.E.D.


[ 3.2 0.1333330000000008 ]
allzero-single-rule
```

Much better. Again, a trivial proof.

Now we get to a hard but crucial lemma. When a program is iterating through an array, clearing each element to zero, we will need to be able to show that clearing each additional element extends the **allzero** property of the array. This will require an inductive proof.

```
-> (prove-lemma allzero-extend-upward-rule
             (rewrite)
             (implies (and (arrayp! a)
                           (numberp i)
                           (numberp j)
                           (allzero a i j)
                           (equal (selecta! a (add1 j)) 0))
                      (allzero a i (add1 j))))
```

Turning the problem over to the theorem prover...

Call the conjecture *1.

Let us appeal to the induction principle. The recursive terms in the conjecture suggest four inductions. They merge into two likely candidate inductions. However, only one is unflawed. We will induct according to the following scheme:

```
(and (implies (lessp j i) (p a i j))
     (implies (and (leq i j) (p a (add1 i) j))
              (p a i j))).
```

Linear arithmetic informs us that the measure (difference (add1 j) i) decreases according to the well-founded relation lessp in each induction step of the scheme. The above induction scheme generates three new formulas:

```
Case 3. (implies (and (lessp j i)
                      (arrayp! a)
                      (numberp i)
                      (numberp j)
                      (allzero a i j)
                      (equal (selecta! a (add1 j)) 0))
                 (allzero a i (add1 j))),
```

which simplifies, appealing to the lemmas allzero-void-rule and sub1-add1, and unfolding allzero and lessp, to four new formulas:

```
Case 3.4.
        (implies (and (lessp j i)
                      (arrayp! a)
                      (numberp i)
                      (numberp j)
                      (equal (selecta! a (add1 j)) 0)
                      (leq (sub1 i) j))
                 (allzero a (add1 i) (add1 j))),
```

which again simplifies, using linear arithmetic, to:

```
        (implies (and (lessp j (plus 1 j))
                      (arrayp! a)
                      (numberp (plus 1 j))
                      (numberp j)
                      (equal (selecta! a (add1 j)) 0)
                      (leq (sub1 (plus 1 j)) j))
                 (allzero a
                          (add1 (plus 1 j))
```

```
                        (add1 j))).
```

But this again simplifies, using linear arithmetic, rewriting with plus-1, sub1-add1, allzero-void-rule, and x-not-less-than-x, and unfolding the definitions of lessp, plus, numberp, add1, and sub1, to:

```
      t.
```

Case 3.3.

```
      (implies (and (lessp j i)
                    (arrayp! a)
                    (numberp i)
                    (numberp j)
                    (equal (selecta! a (add1 j)) 0)
                    (leq (sub1 i) j))
               (equal (selecta! a i) 0)).
```

This again simplifies, using linear arithmetic, to:

```
      (implies (and (lessp j (plus 1 j))
                    (arrayp! a)
                    (numberp (plus 1 j))
                    (numberp j)
                    (equal (selecta! a (add1 j)) 0)
                    (leq (sub1 (plus 1 j)) j))
               (equal (selecta! a (plus 1 j)) 0)).
```

But this again simplifies, applying the lemmas plus-1, sub1-add1, and x-not-less-than-x, and unfolding the functions lessp, plus, numberp, add1, sub1, and equal, to:

```
      t.
```

Case 3.2.

```
      (implies (and (lessp j i)
                    (arrayp! a)
                    (numberp i)
                    (numberp j)
                    (equal (selecta! a (add1 j)) 0)
                    (equal i 0))
               (allzero a (add1 i) (add1 j))),
```

which again simplifies, using linear arithmetic, to:

```
      t.
```

```
  Case 3.1.
          (implies (and (lessp j i)
                        (arrayp! a)
                        (numberp i)
                        (numberp j)
                        (equal (selecta! a (add1 j)) 0)
                        (equal i 0))
                   (equal (selecta! a i) 0)),
```

which again simplifies, using linear arithmetic, to:

```
          t.

Case 2. (implies (and (leq i j)
                      (not (allzero a (add1 i) j))
                      (arrayp! a)
                      (numberp i)
                      (numberp j)
                      (allzero a i j)
                      (equal (selecta! a (add1 j)) 0))
                 (allzero a i (add1 j))),
```

which simplifies, opening up allzero, to:

```
        t.

Case 1. (implies (and (leq i j)
                      (allzero a (add1 i) (add1 j))
                      (arrayp! a)
                      (numberp i)
                      (numberp j)
                      (allzero a i j)
                      (equal (selecta! a (add1 j)) 0))
                 (allzero a i (add1 j))),
```

which simplifies, applying sub1-add1, and opening up the definitions of allzero, lessp, and equal, to:

```
        t.
```

That finishes the proof of *1. Q.E.D.

```
[ 29.84999999999998 1.966667000000014 ]
```

In 30 seconds, an inductive proof, produced without manual intervention. This is Boyer and Moore's great accomplishment. It took them seven years to write the program that does this. Note that our earlier lemma **allzero-void-rule** was used in the proof; we are teaching the prover more and more facts about **allzero.**

Now a seemingly simple but non-trivial property; storing into the array outside the bounds of **allzero** does not affect the **allzero** property.

```
-> (prove-lemma allzero-unchanged-1-rule
            (rewrite)
            (implies (and (numberp i)
                          (numberp j)
                          (arrayp! a)
                          (allzero a i j)
                          (numberp x)
                          (or (lessp x i) (lessp j x)))
                     (allzero (storea! a x v) i j)))
```

The prover takes over...

This conjecture simplifies, opening up the function or, to two new goals:

```
Case 2. (implies (and (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (allzero a i j)
                      (numberp x)
                      (lessp x i))
                 (allzero (storea! a x v) i j)),
```

which we will name *1.

```
Case 1. (implies (and (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (allzero a i j)
                      (numberp x)
                      (lessp j x))
                 (allzero (storea! a x v) i j)),
```

which we would usually push and work on later by induction. But if we must use induction to prove the input conjecture, we prefer to induct on the original formulation of the problem. Thus we will disregard all that we have previously done, give the name *1 to the original input, and work on it.

So now let us consider:

```
(and (implies (and (numberp i)
                   (numberp j)
                   (arrayp! a)
                   (allzero a i j)
                   (numberp x)
                   (lessp j x))
              (allzero (storea! a x v) i j))
     (implies (and (numberp i)
                   (numberp j)
                   (arrayp! a)
                   (allzero a i j)
                   (numberp x)
                   (lessp x i))
              (allzero (storea! a x v) i j))),
```

which we named *1 above. We will appeal to induction. The

recursive terms in the conjecture suggest 12 inductions.
They merge into three likely candidate inductions. However,
only one is unflawed. We will induct according to the following
scheme:

```
      (and (implies (lessp j i) (p a x v i j))
           (implies (and (leq i j) (p a x v (add1 i) j))
                    (p a x v i j))).
```

Linear arithmetic informs us that the measure (difference
(add1 j) i) decreases according to the well-founded relation
lessp in each induction step of the scheme. The above induction
scheme produces the following seven new goals:

```
Case 7. (implies (and (lessp j i)
                      (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (allzero a i j)
                      (numberp x)
                      (lessp j x))
                 (allzero (storea! a x v) i j)).
```

  This simplifies, applying the lemmas allzero-void-rule and
  store-is-proper, to:

```
        t.
```

```
Case 6. (implies (and (leq i j)
                      (not (allzero a (add1 i) j))
                      (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (allzero a i j)
                      (numberp x)
                      (lessp j x))
                 (allzero (storea! a x v) i j)).
```

  This simplifies, unfolding the definition of allzero, to:

```
        t.
```

```
Case 5. (implies (and (leq i j)
                      (allzero (storea! a x v) (add1 i) j)
                      (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (allzero a i j)
```

```
                    (numberp x)
                    (lessp j x))
               (allzero (storea! a x v) i j)).
```

This simplifies, rewriting with select-of-store, and unfolding the function allzero, to:

```
      (implies (and (leq i j)
                    (allzero (storea! a x v) (add1 i) j)
                    (numberp i)
                    (numberp j)
                    (arrayp! a)
                    (allzero a (add1 i) j)
                    (equal (selecta! a i) 0)
                    (numberp x)
                    (lessp j x)
                    (equal x i))
               (equal v 0)).
```

This again simplifies, trivially, to:

```
      t.
```

```
Case 4. (implies (and (lessp j i)
                      (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (allzero a i j)
                      (numberp x)
                      (lessp x i))
                 (allzero (storea! a x v) i j)).
```

This simplifies, applying allzero-void-rule and store-is-proper, to:

```
      t.
```

```
Case 3. (implies (and (leq i j)
                      (leq x j)
                      (leq (add1 i) x)
                      (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (allzero a i j)
                      (numberp x)
                      (lessp x i))
```

```
                (allzero (storea! a x v) i j)),
```

which simplifies, using linear arithmetic, to:

```
      t.
```

```
Case 2. (implies (and (leq i j)
                      (not (allzero a (add1 i) j))
                      (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (allzero a i j)
                      (numberp x)
                      (lessp x i))
                (allzero (storea! a x v) i j)),
```

which simplifies, unfolding the definition of allzero, to:

```
      t.
```

```
Case 1. (implies (and (leq i j)
                      (allzero (storea! a x v) (add1 i) j)
                      (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (allzero a i j)
                      (numberp x)
                      (lessp x i))
                (allzero (storea! a x v) i j)),
```

which simplifies, rewriting with the lemma select-of-store, and opening up the function allzero, to:

```
      (implies (and (leq i j)
                    (allzero (storea! a x v) (add1 i) j)
                    (numberp i)
                    (numberp j)
                    (arrayp! a)
                    (allzero a (add1 i) j)
                    (equal (selecta! a i) 0)
                    (numberp x)
                    (lessp x i)
                    (equal x i))
               (equal v 0)).
```

However this again simplifies, using linear arithmetic, to:

```
          t.


       That finishes the proof of *1.  Q.E.D.


[ 79.36666600000002 2.65000099999999 ]
allzero-unchanged-1-rule
```

An unexpectedly difficult proof; the prover went down a blind alley, backed up, started over, began induction, performed a case analysis, and found a proof.

For our next rule, we prove that storing zero into an array does not cause the **allzero** predicate to become false.

```
-> (prove-lemma allzero-unchanged-2-rule
                (rewrite)
                (implies (and (allzero a i j)
                              (arrayp! a)
                              (numberp i)
                              (numberp j)
                              (numberp x))
                         (allzero (storea! a x 0) i j)))
```

The prover replies:

Call the conjecture *1.

We will appeal to induction. Four inductions are suggested by terms in the conjecture. They merge into two likely candidate inductions. However, only one is unflawed. We will induct according to the following scheme:

```
(and (implies (lessp j i) (p a x i j))
     (implies (and (leq i j) (p a x (add1 i) j))
              (p a x i j))).
```

Linear arithmetic informs us that the measure (difference (add1 j) i) decreases according to the well-founded relation lessp in each induction step of the scheme. The above induction scheme produces the following three new goals:

```
Case 3. (implies (and (lessp j i)
                      (allzero a i j)
                      (arrayp! a)
                      (numberp i)
                      (numberp j)
                      (numberp x))
                 (allzero (storea! a x 0) i j)).
```

This simplifies, rewriting with allzero-void-rule and store-is-proper, to:

```
t.
```

```
Case 2. (implies (and (leq i j)
                      (not (allzero a (add1 i) j))
                      (allzero a i j)
                      (arrayp! a)
                      (numberp i)
                      (numberp j)
                      (numberp x))
                 (allzero (storea! a x 0) i j)),
```

which simplifies, opening up allzero, to:

```
t.
```

```
Case 1. (implies (and (leq i j)
                      (allzero (storea! a x 0) (add1 i) j)
                      (allzero a i j)
                      (arrayp! a)
                      (numberp i)
                      (numberp j)
```

```
                    (numberp x))
               (allzero (storea! a x 0) i j)),
```

which simplifies, applying select-of-store, and opening up the functions allzero and equal, to:

```
      t.


   That finishes the proof of *1.  Q.E.D.


[ 20.23333099999997 1.10000200000001 ]
allzero-unchanged-2-rule
```

That one wasn't too hard.

Finally, the rule that lets us get some payoff from using **allzero** in a program verification; we show that if **allzero** is true for A from I to J, then for any element X between I and J, then A[X] = 0.

```
-> (prove-lemma allzero-select-rule
            (rewrite)
            (implies (and (allzero a i j)
                          (numberp i)
                          (numberp j)
                          (arrayp! a)
                          (numberp x)
                          (leq x j)
                          (leq i x))
                     (equal (selecta! a x) 0)))
```

Over to the prover.

```
WARNING:  Note that allzero-select-rule contains the free
variables j and i which will be chosen by instantiating
the hypothesis (allzero a i j).
```

Here the prover grumbles at us; our rule is not well chosen according to its built-in ideas as to what an efficient rewrite lemma is. Rules of this type may slow down the prover in later proofs. In this case, though, there is no better way to state this rule. The theorem prover proceeds; this was only a WARNING. There is no risk to soundness here.

Name the conjecture *1.

Let us appeal to the induction principle. There are seven plausible inductions. They merge into three likely candidate inductions. However, only one is unflawed. We will induct according to the following scheme:

```
 (and (implies (lessp j i) (p a x i j))
      (implies (and (leq i j) (p a x (add1 i) j))
               (p a x i j))).
```

Linear arithmetic establishes that the measure (difference (add1 j) i) decreases according to the well-founded relation lessp in each induction step of the scheme. The above induction scheme generates three new formulas:

```
Case 3. (implies (and (lessp j i)
                      (allzero a i j)
                      (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (numberp x)
                      (leq x j)
                      (leq i x))
                 (equal (selecta! a x) 0)),
```

which simplifies, using linear arithmetic, to:

t.

```
Case 2. (implies (and (leq i j)
                      (not (allzero a (add1 i) j))
                      (allzero a i j)
                      (numberp i)
                      (numberp j)
                      (arrayp! a)
                      (numberp x)
                      (leq x j)
                      (leq i x))
                 (equal (selecta! a x) 0)),
```

which simplifies, unfolding allzero, to:

t.

```
Case 1. (implies (and (leq i j)
                      (lessp x (add1 i))
```

```
                    (allzero a i j)
                    (numberp i)
                    (numberp j)
                    (arrayp! a)
                    (numberp x)
                    (leq x j)
                    (leq i x))
               (equal (selecta! a x) 0)),
```

which simplifies, using linear arithmetic, to:

```
      (implies (and (leq i j)
                    (lessp i (add1 i))
                    (allzero a i j)
                    (numberp i)
                    (numberp j)
                    (arrayp! a)
                    (numberp i)
                    (leq i j)
                    (leq i i))
               (equal (selecta! a i) 0)).
```

But this again simplifies, applying sub1-add1, and opening up lessp, numberp, equal, and allzero, to:

```
      (implies (and (equal i 0)
                    (allzero a 0 j)
                    (numberp j)
                    (arrayp! a))
               (equal (selecta! a 0) 0)),
```

which again simplifies, opening up the definitions of add1, lessp, equal, and allzero, to:

```
      t.
```

```
   That finishes the proof of *1.  Q.E.D.
```

```
[ 15.0 1.266665999999987 ]
allzero-select-rule
```

Success. We now have a set of rules which will allow us to use **allzero** in a verification and to use it in most of the reasonable ways to use such a predicate. Note that this

approach will work for any predicate based on properties of individual array elements. **Allzero** is a simple example.

We are done proving; it is time to make a library file for use with the Verifier (or for later use with **note-lib** in case we need some more lemmas for our verification).

```
-> (make-lib 'allzero)
(%$unopenedport %$unopenedport)
```

The files *allzero.lib* and *allzero.lisp* have now been created in the current directory. Together these constitute our new knowledge base. We are now ready to leave the Rule Builder.

```
-> (exit)
```

This returns us to the UNIX shell. At this point, we can put our new rules in the working directory created by the Verifier for the program **example6.pf** by using the **putrules** utility program, which extracts all the needed information from a Rule Builder database and puts it into a much more compact file which the Verifier can use.

```
% putrules allzero.lib example6_d
```

Putrules runs and prints some messages. This is just a format translation; nothing profound is going on here.

```
Processing database allzero.lib
Installing new database in example6_d
```

We can now rerun our verification.

```
% pasver example6.pf
```

and the verifier prints

```
Pass 1:
Pass 2:
Pass 3:

Verifying example6
No errors detected
```

so our verification is a success.

7.6 Additional output from the verifier

To show what the Verifier actually does with the information provided through rules, we show some of the diagnostic output the Verifier produces for the use of those building and testing rules. When a verification is unsuccessful, it is usually best to try to fix the problem by looking at the source program. There are, though, additional outputs available from the Verifier for dealing with difficult problems.

This section is not intended to give a real understanding of how the Verifier works. There is an internal documentation manual for the Verifier, and the sections of that manual entitled **Icode to Jcode Translator** and **Verification Condition Generator** cover the generation of verification conditions and the internal file formats in much greater detail.

7.6.1 The logging file

The output below all appears in the file **p3-vcs** in the Verifier's scratch directory **programname-dvcg flag is set on the call to pasver.)**

7.6.1.1 The rule listing

The first part of this file is the list of rules found in the **ruledatabase** file.

```
allzero-extend-upward-rule --
    Usable on conclusions only, free variables (A I J g00002)
Trigger pattern sequence: ((allzero A I g00002) (allzero A I J))

arrayp!(A)
and numberp!(I)
and numberp!(J)
and allzero(A,I,J)
and (A[addn!(J,1)] = 0)
implies
  allzero(A,I,addn!(J,1))
```

We remember this rule from the Rule Builder session. Here, the rules appear in infix form, in Pascal-like notation. Operators which have no Pascal-F source representation appear as function calls. These names all end with ‘‘!’’ to avoid interference with user-defined functions. Note that numberp has become **numberp!,** and (add1 J) has become **addn!(J,1),** where **addn!** is the plus function (natural number add) from the Rule

Builder.

The list of free variables lists all the terms which must be bound when the rule is instantiated. The dummy name **g00002** is a placeholder for the **addn!** term in the conclusion.

The **trigger pattern sequence** shows when the rule will be applied. The Verifier will look at the verification condition, and will try to match the first pattern in the sequence. Once it matches, the variables in that pattern are bound and the Verifier begins looking for the second pattern in the sequence. The message ‘‘usable on conclusions only’’ indicates that the first pattern is general enough (containing only one function symbol) that applying the rule everywhere in every way that **allzero** appeared would slow down the system too much. So this rule will only be applied when **allzero** appears in a proof goal, that is, something written in an **ASSERT, STATE, SUMMARY, ENTRY, exit,** or **INVARIANT,** or (not possible in this case) in an internally generated requirement used to insure subscripts within range or arithmetic results within bounds. Within this limitation, rules are applied in every possible combination of ways, but only one deep, so if a proof can be found in one step with the given rules, it will be found.

```
allzero-select-rule --
    Usable on conclusions only, free variables (A I J X)
Trigger pattern sequence: ((selecta! A X) (allzero A I J))

allzero(A,I,J)
and numberp!(I)
and numberp!(J)
and arrayp!(A)
and numberp!(X)
and not gtn!(X,J)
and not gtn!(I,X)
implies
  A[X] = 0
```

Note that A[X] is **(selecta! A X)** in the pattern. The function **gtn!** is the greater than operator for the natural numbers.

```
allzero-single-rule --
    Usable on conclusions only, free variables (A I J)
Trigger pattern sequence: ((allzero A I J))

arrayp!(A) and numberp!(I) and numberp!(J) and (I = J) and (A[I] = 0)
implies
  allzero(A,I,J)
```

```
allzero-unchanged-1-rule --
    Usable anywhere, free variables (A I J V X)
Trigger pattern sequence: ((allzero (storea! A X V) I J))

numberp!(I)
and numberp!(J)
and arrayp!(A)
and allzero(A,I,J)
and numberp!(X)
and (gtn!(I,X) or gtn!(X,J))
implies
  allzero(<A,X,V>,I,J)
```

Note that this rule is usable anywhere. Because there are nested function symbols in the pattern, we don't expect to see too many places where this rule could be applied without purpose. Also note the appearance of **<A,X,V>** which is the infix form of (storea! A X V).

```
allzero-unchanged-2-rule --
    Usable anywhere, free variables (A I J X)
Trigger pattern sequence: ((allzero (storea! A X 0) I J))

allzero(A,I,J)
and arrayp!(A)
and numberp!(I)
and numberp!(J)
and numberp!(X)
implies
  allzero(<A,X,0>,I,J)
```

```
allzero-void-rule --
    Usable on conclusions only, free variables (A I J)
Trigger pattern sequence: ((allzero A I J))

arrayp!(A) and numberp!(I) and numberp!(J) and gtn!(I,J)
implies
  allzero(A,I,J)
```

The following rules are part of the standard database, and are used for proving definedness.

```
arraytrue-extend-upward-rule --
    Usable on conclusions only, free variables (A I J g00005)
Trigger pattern sequence: ((arraytrue! A I g00005) (arraytrue! A I J)

(arraytrue!(A,I,J) = true) and (alltrue!(A[addn!(J,1)]) = true)
implies
  arraytrue!(A,I,addn!(J,1)) = true



arraytrue-single-rule --
    Usable on conclusions only, free variables (A I)
Trigger pattern sequence: ((selecta! A I))

arraytrue!(A,I,I) = true = alltrue!(A[I]) = true



arraytrue-unchanged-rule --
    Usable anywhere, free variables (A I J V X)
Trigger pattern sequence: ((arraytrue! (storea! A X V) I J))

numberp!(X)
and numberp!(I)
and numberp!(J)
and arrayp!(A)
and (arraytrue!(A,I,J) = true)
and (gtn!(I,X) or gtn!(X,J))
implies
  arraytrue!(<A,X,V>,I,J) = true
```

```
arraytrue-void-rule --
    Usable on conclusions only, free variables (A I J)
Trigger pattern sequence: ((arraytrue! A I J))

gtn!(I,J) implies arraytrue!(A,I,J)
```

7.6.1.2 The verification trace

Now we go on to the verification goals themselves. If a verification condition had failed, it would have been printed, but all these succeeded, so only the goal and path appear. This is exactly the text that would appear as an error message if the proof failed.

```
Verification condition for {example6.pf:18} table1[(j - 1) + 1] = 0
    (ASSERT assertion)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} FOR loop never entered
    Tried arraytrue-single-rule.
VC #1 proved in 1.00 seconds.
```

This one took one second, and **arraytrue-single-rule** was successfully pattern-matched, although it is irrelevant in this case. The message ‘‘Tried’’ indicates only that the rule could have been applied, not that it actually was.

```
Verification condition for {example6.pf:18} table1[(j - 1) + 1] = 0
    (ASSERT assertion)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} FOR loop exit
    Tried allzero-select-rule.
    Tried arraytrue-single-rule.
VC #2 proved in 2.70 seconds.
```

Here there is some statement about definedness in the hypothesis of the verification condition which triggered the built-in rule **arraytrue-single-rule.**

```
Verification condition for {example6.pf:16} allzero(table1,1,100)
    (ASSERT assertion)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} FOR loop never entered
    Tried allzero-extend-upward-rule.
    Tried allzero-single-rule.
    Tried allzero-void-rule.
VC #3 proved in 0.63 seconds.
```

We do not get any useful information about which rule did it, if any.

```
Verification condition for {example6.pf:16} allzero(table1,1,100)
    (ASSERT assertion)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} FOR loop exit
    Tried allzero-extend-upward-rule 2 times.
    Tried allzero-single-rule.
    Tried allzero-void-rule.
VC #4 proved in 1.00 seconds.
```

```
Verification condition for {example6.pf:11} i <= 99
    (FOR loop count)
Path:
    {example6.pf:11} Start of "example6"
VC #5 proved in 0.75 seconds.
```

```
Verification condition for {example6.pf:14} allzero(table1,1,i)
    (STATE assertion)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:15} Back to top of FOR loop
    Tried allzero-unchanged-2-rule 2 times.
    Tried allzero-unchanged-1-rule 2 times.
    Tried allzero-extend-upward-rule 2 times.
    Tried allzero-single-rule.
    Tried allzero-void-rule.
VC #6 proved in 28.81 seconds.
```

This is the inductive case around the loop, the hard one. It took 28 seconds.

```
Verification condition for {example6.pf:14} allzero(table1,1,i)
    (STATE assertion)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} Enter FOR loop
    Tried allzero-unchanged-2-rule 2 times.
    Tried allzero-unchanged-1-rule 2 times.
    Tried allzero-extend-upward-rule 2 times.
    Tried allzero-single-rule.
    Tried allzero-void-rule.
VC #7 proved in 2.45 seconds.
```

```
Verification condition for {example6.pf:13} allzero(table1,1,i - 1)
    (ASSERT assertion)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:15} Back to top of FOR loop
    Tried allzero-unchanged-2-rule.
    Tried allzero-unchanged-1-rule.
    Tried allzero-extend-upward-rule.
    Tried allzero-single-rule.
    Tried allzero-void-rule.
VC #8 proved in 9.25 seconds.
```

```
Verification condition for {example6.pf:13} allzero(table1,1,i - 1)
    (ASSERT assertion)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} Enter FOR loop
    Tried allzero-unchanged-2-rule.
    Tried allzero-unchanged-1-rule.
    Tried allzero-extend-upward-rule.
    Tried allzero-single-rule.
    Tried allzero-void-rule.
VC #9 proved in 2.03 seconds.
```

```
Verification condition for {example6.pf:12} i - 1 <= 99
    (subscript check for "table1" 1..100)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:15} Back to top of FOR loop
VC #10 proved in 0.30 seconds.
```

This is an internally-generated proof goal, a subscript check.

```
Verification condition for {example6.pf:12} i - 1 <= 99
    (subscript check for "table1" 1..100)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} Enter FOR loop
VC #11 proved in 0.26 seconds.
```

```
Verification condition for {example6.pf:12} i - 1 >= 0
    (subscript check for "table1" 1..100)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:15} Back to top of FOR loop
VC #12 proved in 0.31 seconds.
```

```
Verification condition for {example6.pf:12} i - 1 >= 0
    (subscript check for "table1" 1..100)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} Enter FOR loop
VC #13 proved in 0.26 seconds.
```

```
Verification condition for {example6.pf:12} "i" is defined
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:15} Back to top of FOR loop
VC #14 proved in 0.08 seconds.
```

This is a test to make sure the variable **i** was defined at line 12. These usually are very fast to prove.

```
Verification condition for {example6.pf:12} "i" is defined
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:11} Enter FOR loop
VC #15 proved in 0.06 seconds.
```

That is the summary of the verification. When a verification is unsuccessful, and it is is not clear why, examination of this file can be quite useful. Remember that this file is not generated unless requested, and generating it does slow down the verification by about 20-40%.

7.6.1.3 What verification conditions look like

It is not usually necessary for the user to look at verification conditions, but when difficulties are encountered it can sometimes be useful. When using a compiler, it is sometimes useful to turn on a listing of generated object code. The equivalent listing for a verifier is the listing of verification conditions. We thus provide an explanation as to how to read a verification condition.

Had a verification condition failed, in the above example, we would see, in this log of verification conditions, the verification condition itself. If we force some errors by trying to run the verification above without any rules about **allzero** available, we would find the information below in the log.

```
Verification condition for {example6.pf:14} allzero(table1,1,i)
    (STATE assertion)
Path:
    {example6.pf:11} Start of "example6"
    {example6.pf:15} Back to top of FOR loop

(TEMP4__v01 = 1)
and (TEMP5__v01 = 100)
and true
and (i_4v03 <= TEMP5__v01)
and allzero(table1_2v02,1,i_4v03)
and (i_4v03 >= TEMP4__v01)
and (i_4v03 < TEMP5__v01)
and (i_4v03 <= 99)
and (i_4v04 = i_4v03 + 1)
and (i_4v04 - 1 >= 0)
and (i_4v04 - 1 <= 99)
and (table1_2v03 = (table1_2v02[(i_4v04 - 1) + 1] := 0))
and allzero(table1_2v03,1,i_4v04 - 1)
implies
  allzero(table1_2v03,1,i_4v04)

VC #6 FAILED in 1.48 seconds.
```

Here, we see the actual verification condition to be proven. This verification condition is for the path around the loop, with the proof goal of **allzero(table1,1,i)** in the STATE statement.

Verification conditions are always of the form ‘‘big conjunction implies proof goal’’. The proof goal is always the term printed after the implication. The terms in the hypothesis are generated by backwards tracing through the program, examining each statement along the indicated path. The details of how this is done are beyond the scope of this manual, but generally follow the backwards-tracing approach of Floyd, Manna, and others. The basic idea is that programs are converted to formulas by tracing backwards through each statement, using a new variable name for each variable every time its value is changed. For example, the Pascal-F statements

```
x := 1;
x := x + 1;
assert(x = 2);
```

would generate a verification condition of the form

```
    x_1v01 = 1
and x_1v02 = (x_1v01 + 1)
implies
    x_1v02 = 2
```

Note that **x_1v01** and **x_1v02** represent values of **x** at different points in the program. In our **allzero** example, the variable names **i** and **table1** appear here in modified form; the **table1_2v03** string is the variable **table1** after the third assignment to it. The names are actually constructed by taking the user's name of the variable, adding a delimiter and a variable serial number, (so that variables with the same name but in different scopes are made unique) and adding a **v** or a **d,** where a **v** indicates that the **value** of the variable is being referred to, and **d** indicates that the **definedness part** of the variable is being referenced. Finally, a two digit suffix indicating, as in the little example with **x** above, which value of the variable we are referring to, is added to the name. the code.

The **v** or **d** component deserves some extra discussion. Definedness of variables is handled by a convenient fiction. We pretend that associated with each variable there is a definedness flag, set to true when any value is assigned to the variable and tested at every reference to the variable. We then try to prove that the flag is always true at every reference. For an array or record our definedness part will be an array or record with all-Boolean elements or lowest-level fields. When the Verifier generates a verification condition about the definedness of a variable, it constructs a name using the **d** letter.

Not shown in the verification condition is the type information. The theorem prover has available to it all the information in the type declarations of the Pascal-F program. For example, if the declaration

```
var i: 0..100;
```

appeared in a Pascal-F program, the theorem prover would be able to prove

```
true
implies
    i_1v01 <= 100
```

without any difficulty. We can assume that all variables stay within their type because we generate proof goals for the bounds of every variable at every assignment to that variable. Since we also check at every reference to every variable that the variable has been initialized (defined) we are thus safe in assuming that variables stay within their types.

7.6.2 The diagnostics file

The file **p3-diags** contains any error messages produced during the verification. This is useful if messages scroll by on a CRT terminal and are thereby lost. Only messages from

pass 3 appear in this file, but pass 3 is where all the time goes.

7.6.3 The rule data base file

The rule data base file **ruledatabase** is created by the **putrules** utility and read by the Verifier. It contains the rule data base to be used for the current verification.

7.6.4 The history file

The history file **history** contains the entire intermediate code for each program unit previously verified successfully. On reverification attempts, if the newly generated intermediate code matches that found in the history file, verification of that program unit is skipped. Removal of this file will force a complete rerun of the verification. Note that **putrules** will remove this file if a new database is used with different **defn** entries for some previously used **defn** definition.

8. The formal theory of the Rule Builder

The theory built into the Rule Builder is the Boyer-Moore theory of the natural numbers, plus the definitions and lemmas given in this chapter. The notation is that of **A Computational Logic.** Note that everything here has been proven by the Boyer-Moore prover.

8.1 The theory

We begin by defining the notion of Boolean value.

```
Definition.
(booleanp! X)
  =
(or (equal X (true))
   (equal X (false)))
```

The next step is to add a large batch of carefully chosen facts about the natural numbers.

```
Theorem.  equal-lessp:
(equal (equal (lessp X Y) Z)
      (if (lessp X Y)
         (equal t Z)
         (equal f Z)))

Theorem.  associativity-of-plus:
(equal (plus (plus X Y) Z)
      (plus X (plus Y Z)))

Theorem.  equal-times-0:
(equal (equal (times X Y) 0)
      (or (zerop X) (zerop Y)))

Theorem.  commutativity2-of-plus:
(equal (plus X (plus Y Z))
      (plus Y (plus X Z)))
```

```
Theorem.  commutativity-of-times:
(equal (times X Y) (times Y X))

Theorem.  distributivity-of-times-over-plus:
(equal (times X (plus Y Z))
       (plus (times X Y) (times X Z)))

Theorem.  plus-0:
(equal (plus X 0) (fix X))

Theorem.  plus-1:
(implies (numberp X)
         (equal (plus 1 X) (add1 X)))

Theorem.  x-not-less-than-x:
(equal (lessp X X) f)

Theorem.  times-0:
(equal (times X 0) 0)

Theorem.  plus-non-numberp:
(implies (not (numberp Y))
         (equal (plus X Y) (fix X)))

Theorem.  times-non-numberp:
(implies (not (numberp Y))
         (equal (times X Y) 0))

Theorem.  associativity-of-times:
(equal (times (times X Y) Z)
       (times X (times Y Z)))

Theorem.  commutativity2-of-times:
(equal (times X (times Y Z))
       (times Y (times X Z)))

Theorem.  plus-add1:
(equal (plus X (add1 Y))
       (if (numberp Y)
           (add1 (plus X Y))
           (add1 X)))

Theorem.  times-add1:
(equal (times X (add1 Y))
       (if (numberp Y)
           (plus X (times X Y))
           (fix X)))
```

```
Theorem.  commutativity-of-plus:
(equal (plus X Y) (plus Y X))

Theorem.  plus-equal-0:
(equal (equal (plus A B) 0)
       (and (zerop A) (zerop B)))

Theorem.  plus-cancellation:
(equal (equal (plus A B) (plus A C))
       (equal (fix B) (fix C)))

Theorem.  plus-right-id2:
(implies (not (numberp Y))
         (equal (plus X Y) (fix X)))

Theorem.  monotonicity-of-plus-1:
(implies (and (numberp A)
              (numberp B)
              (numberp C))
         (equal (lessp (plus A B) (plus A C))
                (lessp B C)))

Theorem.  difference-x-x:
(equal (difference X X) 0)

Theorem.  difference-plus-1:
(equal (difference (plus X Y) X)
       (fix Y))

Theorem.  difference-plus-2:
(equal (difference (plus Y X) X)
       (fix Y))

Theorem.  equal-difference-0:
(equal (equal 0 (difference X Y))
       (not (lessp Y X)))

Theorem.  zero-difference:
(implies (lessp A B)
         (equal (difference A B) 0))

Theorem.  plus-difference3:
(equal (difference (plus X Y) (plus X Z))
       (difference Y Z))
```

```
Theorem.  monotonicity-of-difference-1:
(implies (and (numberp V)
              (numberp Y)
              (numberp Z)
              (not (lessp Z V))
              (not (lessp Y V)))
         (equal (lessp (difference Y V)
                       (difference Z V))
                (lessp Y Z)))

Theorem.  monotonicity-of-difference-2:
(implies (and (numberp V)
              (numberp Y)
              (numberp Z)
              (lessp Z V)
              (lessp Y V))
         (equal (lessp (difference V Z)
                       (difference V Y))
                (lessp Y Z)))

Theorem.  monotonicity-of-difference-3:
(implies (and (numberp W)
              (numberp V)
              (numberp X)
              (not (lessp X W))
              (not (lessp X V)))
         (equal (lessp (difference X V)
                       (difference X W))
                (lessp W V)))

Theorem.  times-zero:
(equal (times X 0) 0)

Theorem.  distributivity-of-times-over-difference:
(equal (times X (difference Y Z))
       (difference (times X Y) (times X Z)))

Theorem.  monotonicity-of-times-1:
(implies (and (numberp X)
              (numberp Y)
              (numberp Z)
              (not (zerop X)))
         (equal (not (lessp (times X Y) (times X Z)))
                (not (lessp Y Z))))
```

```
Theorem.  monotonicity-of-times-3:
(implies (and (numberp A)
              (numberp B)
              (numberp C)
              (not (equal C 0)))
         (equal (lessp (times C A) (times C B))
                (lessp A B)))

Theorem.  monotonicity-of-times-by-twos:
(implies (and (lessp X Y) (lessp Z W))
         (lessp (times X Z) (times Y W)))

Theorem.  remainder-x-x:
(equal (remainder X X) 0)

Theorem.  remainder-quotient:
(equal (plus (remainder X Y)
             (times Y (quotient X Y)))
       (fix X))

Theorem.  remainder-quotient-elim:
(implies (and (not (zerop Y)) (numberp X))
         (equal (plus (remainder X Y)
                      (times Y (quotient X Y)))
                X))

Theorem.  remainder-non-numeric:
(implies (not (numberp X))
         (equal (remainder Y X) (fix Y)))

Theorem.  remainder-wrt-1:
(equal (remainder Y 1) 0)

Theorem.  quotient-times:
(equal (quotient (times Y X) Y)
       (if (zerop Y) 0 (fix X)))

Theorem.  monotonicity-of-times:
(implies (and (numberp X)
              (numberp Y)
              (numberp Z)
              (not (lessp Y Z)))
         (equal (lessp (times X Y) (times X Z))
                f))
```

We now add an object called undefined which will be needed in the definition of arrays. This has nothing to do with the Verifier's proofs of definedness; it is just a default object

introduced to make **selecta!** a total function.

Shell Definition.
Add the shell undefined-object of zero arguments with
bottom object undefined,
recognizer undefinedp,
accessors,
and default values.

The definition of arrays is constructive. An array is actually represented as an ordered list of subscript-value pairs.

Shell Definition.
Add the shell array-shell of three arguments with
bottom object empty-array,
recognizer array-recognizer,
accessors array-elt-value, array-elt-subscript, and array-prev,
type restrictions (none-of), (one-of numberp), and:
(one-of array-recognizer)
and default values undefined, zero, and empty.array.

The predicate **arrayp!** is true only if an array is a valid ordered list of pairs, properly ordered in increasing order of subscript. Note that something is an array only if the subscripts in the list are in ascending order and no value part is UNDEFINED.

```
Definition.
(arrayp! A)
  =
(if
  (array-recognizer A)
  (if (equal A (empty-array))
     t
     (if (or (not (numberp (array-elt-subscript A)))
             (equal (array-elt-value A)
                    (undefined)))
        f
        (if (equal (array-prev A) (empty-array))
           t
           (and (lessp (array-elt-subscript (array-prev A))
                       (array-elt-subscript A))
                (arrayp! (array-prev A))))))
  f)
```

selecta! is the array subscripting function, which searches the list of pairs.

```
Definition.
(selecta! A I)
  =
(if (equal (array-elt-subscript A) I)
   (array-elt-value A)
   (if (equal (array-prev A) (empty-array))
      (undefined)
      (selecta! (array-prev A) I)))
```

storea! is quite complex, since it is actually a routine for inserting into an ordered list. Our check on the validity of this is that we are able to prove all the standard theorems about **selecta!** and **storea!,** which are axioms in the Oppen system.

```
Definition.
(storea! A I V)
  =
(if (and (arrayp! A) (numberp I))
   (if (equal A (empty-array))
      (if (equal V (undefined))
         A
         (array-shell V I (empty-array)))
      (if (equal (array-elt-subscript A) I)
         (if (equal V (undefined))
            (array-prev A)
            (array-shell V I (array-prev A)))
         (if (lessp (array-elt-subscript A) I)
            (if (equal V (undefined))
               A
               (array-shell V I A))
            (array-shell (array-elt-value A)
                     (array-elt-subscript A)
                     (storea! (array-prev A) I V)))))
   (empty-array))
```

The result of **storea!** is shown to be a valid array.

```
Theorem.  store-is-proper:
(equal (arrayp! (storea! A I V)) t)
```

We prove the classic lemmas about **selecta!** and **storea!** This not only shows the validity of our definition of **storea!** , but gives the Rule Builder a set of rules which comprise a decision procedure for our array theory.

```
Theorem.  select-of-store-1:
(implies (and (arrayp! A) (numberp I))
       (equal (selecta! (storea! A I V) I)
            V))
```

```
Theorem.  store-of-select:
(implies (and (arrayp! A) (numberp I))
         (equal (storea! A I (selecta! A I))
                A))

Theorem.  select-of-store-2:
(implies (and (arrayp! A)
              (numberp I)
              (numberp J)
              (not (equal I J)))
         (equal (selecta! (storea! A I V) J)
                (selecta! A J)))

Theorem.  select-of-store:
(implies (and (arrayp! A)
              (numberp I)
              (numberp J))
         (equal (selecta! (storea! A I V) J)
                (if (equal I J) V (selecta! A J))))

Theorem.  store-of-store-1:
(implies (and (arrayp! A) (numberp I))
         (equal (storea! (storea! A I V) I W)
                (storea! A I W)))
```

storer! is the record store function. The Rule Builder does not know about Verifier record structures, but the definition as an undefined function allows the appearance of **storer!** in rules. Of course, the only thing known about it in the Rule Builder is that if the arguments to **storer!** are the same, the result is the same. The Verifier proper has built-in knowledge about **storer!** and **selectr!0** , but that knowledge is type-dependent and cannot be used here.

```
Undefined Function.
(storer! A B C)

Undefined Function.
(selectr! r I)
```

alltrue! is true of an object if and only if all its components have the Boolean value **TRUE.** The Verifier has built-in knowledge about **alltrue!** and, as with the record operators, that knowledge is type-dependent.

```
Undefined Function.
(alltrue! r)
```

Integers are built up by defining a Boyer-Moore shell such that negative numbers are a shell whose negative-guts field contains the natural number for the absolute value. This creates a problem in that there is such a thing as negative zero. This definition of **integerp!** disallows negative zero, and all our operations on the integers never produce

negative zero.

```
Definition.
(integerp! X)
  =
(if (numberp X)
    t
    (if (negativep X)
        (if (zerop (negative-guts X)) f t)
        f))
```

This turns negative zero into positive zero.

```
Definition.
(znormalize X)
  =
(if (negativep X)
    (if (equal (negative-guts X) 0) 0 X)
    X)
```

This is a conversion from a natural number to a negative number which avoids minus zero.

```
Definition.
(zmonus X)
  =
(znormalize (minus X))
```

Unary negation.

```
Definition.
(negi! X)
  =
(if (integerp! X)
    (if (negativep X)
        (negative-guts X)
        (zmonus X))
    0)
```

Integer addition is defined by cases. Proofs about integer arithmetic thus generate extensive case analysis, and due to a limitation of the Boyer-Moore prover it does not help to provide lemmas about nonrecursive definitions. Therefore there are no lemmas in this knowledge base about the integer arithmetic functions. It is quite possible to prove rules about **addi!** and its friends, and it is not usually difficult, but such proofs run slowly.

```
Definition.
(addi! X Y)
  =
(if (negativep X)
   (if (negativep Y)
      (zmonus (plus (negative-guts X)
                    (negative-guts Y)))
      (if (lessp Y (negative-guts X))
         (zmonus (difference (negative-guts X) Y))
         (difference Y (negative-guts X))))
   (if (negativep Y)
      (if (lessp X (negative-guts Y))
         (zmonus (difference (negative-guts Y) X))
         (difference X (negative-guts Y)))
      (plus X Y)))

Definition.
(subi! X Y)
  =
(addi! X (negi! Y))

Definition.
(muli! X Y)
  =
(if (negativep X)
   (if (negativep Y)
      (times (negative-guts X)
             (negative-guts Y))
      (zmonus (times (negative-guts X) Y)))
   (if (negativep Y)
      (zmonus (times X (negative-guts Y)))
      (times X Y)))

Definition.
(divi! X Y)
  =
(if (negativep X)
   (if (negativep Y)
      (quotient (negative-guts X)
                (negative-guts Y))
      (zmonus (quotient (negative-guts X) Y)))
   (if (negativep Y)
      (zmonus (quotient X (negative-guts Y)))
      (quotient X Y)))
```

The integer relational operators are defined by cases.

```
Definition.
(lti! X Y)
  =
(if (negativep X)
   (if (negativep Y)
      (lessp (negative-guts Y)
             (negative-guts X))
      (not (and (equal (negative-guts X) 0)
                (zerop Y))))
   (if (negativep Y) f (lessp X Y)))

Definition.
(gti! X Y)
  =
(lti! Y X)

Definition.
(gei! X Y)
  =
(not (lti! X Y))

Definition.
(lei! X Y)
  =
(not (lti! Y X))
```

zabs is not actually used in rule building, but has been used in producing soundness proofs for the definitions of integer arithmetic.

```
Definition.
(zabs X)
  =
(if (negativep X) (negative-guts X) X)

Definition.
(sign-mult X Y)
  =
(if (equal X 1)
   Y
   (if (equal Y 1) -1 1))

Definition.
(positivep X)
  =
(if (numberp X)
   (if (not (zerop X)) t f)
   f)
```

```
Definition.
(sign X)
  =
(if (numberp X)
   1
   (if (negativep X) -1 0))

Definition.
(switch s X)
  =
(if (equal s 1) X (negi! X))

Definition.
(negative-and-non-zerop X)
  =
(if (negativep X)
   (if (not (zerop (negative-guts X)))
      t f)
   f)

Definition.
(diff-plus-1 X Y)
  =
(difference (add1 Y) X)
```

The **arraytrue!** function is used in showing definedness. **arraytrue!** of **A** is true from **I** to **J** if and only if every element of **A** within the range **I** to **J** is equal to true. The Verifier will crank out **arraytrue!** forms when the user writes

```
defined(A,I,J)
```

or

```
defined(A)
```

where **A** is an array.

```
Definition.
(arraytrue! A I J)
  =
(if (lessp J I)
   t
   (and (equal (alltrue! (selecta! A I)) t)
      (arraytrue! A (add1 I) J)))
```

We have all the obvious rules about **arraytrue!.**

```
Theorem.  arraytrue-void-rule:
(implies (lessp J I)
      (arraytrue! A I J))
```

```
Theorem.  arraytrue-single-rule:
(equal (equal (arraytrue! A I I) t)
     (equal (alltrue! (selecta! A I)) t))

Theorem.  arraytrue-extend-upward-rule:
(implies (and (equal (arraytrue! A I J) t)
              (equal (alltrue! (selecta! A (add1 J)))
                     t))
       (equal (arraytrue! A I (add1 J)) t))

Theorem.  arraytrue-unchanged-rule:
(implies (and (numberp X)
              (numberp I)
              (numberp J)
              (arrayp! A)
              (equal (arraytrue! A I J) t)
              (or (lessp X I) (lessp J X)))
       (equal (arraytrue! (storea! A X V) I J)
              t))

Theorem.  arraytrue-unchanged-2-rule:
(implies (and (numberp X)
              (numberp I)
              (numberp J)
              (arrayp! A)
              (equal (alltrue! V) t)
              (equal (arraytrue! A I J) t))
       (equal (arraytrue! (storea! A X V) I J)
              t))

Theorem.  arraytrue-select-rule:
(implies (and (arraytrue! A I J)
              (numberp I)
              (numberp J)
              (numberp X)
              (not (lessp X I))
              (not (lessp J X)))
       (alltrue! (selecta! A X)))
```

Finally, we have the array construction function. This function is used to construct constant arrays in which all elements are the same. The only use for this function is to construct objects which represent the definedness parts of arrays known to be defined. When an entire array replacement appears in Pascal-f, the definedness part of the array will be set equal to a value built with **arrayconstruct!** in the verification condition.

```
Definition.
(arrayconstruct! V I J)
  =
(if (lessp J I)
   (empty-array)
   (storea! (arrayconstruct! V (add1 I) J)
         I V))

Theorem.  arrayconstruct-is-arrayp:
(arrayp! (arrayconstruct! V I J))

Theorem.  arrayconstruct-select-rule:
(implies (and (numberp I)
              (numberp J)
              (numberp X)
              (not (lessp X I))
              (not (lessp J X)))
         (equal (selecta! (arrayconstruct! V I J) X)
                V))

Theorem.  arrayconstruct-implies-arraytrue-rule:
(implies (and (numberp I)
              (numberp J)
              (equal (alltrue! V) t))
         (equal (arraytrue! (arrayconstruct! V I J)
                            I J)
                t))
```

That is the built-in knowledge base. It takes about two hours to prove.

9. Appendices

9.1 Restrictions and limitations

This release of the system is the first release, and while we have taken extensive precautions against unsoundness we cannot at this time make strong statements about the validity of the verifications. A number of features are unimplemented or implemented with restrictions; however, none of these limitations affect soundness.

9.1.1 Unimplemented features

The system is faithful to this manual except as noted below.

- Fixed point arithmetic is unimplemented.
- The SUMMARY statement is not implemented.
- Arrays with negative lower bounds are prohibited.
- Variant records are unimplemented.
- Arrays with Boolean subscripts (not elements) are prohibited.
- Most set operators are unimplemented.
- The built-in functions of Pascal are unimplemented. The type coercions *chr* and *ord*, along with the Pascal-F named type coercions, are implemented.
- The EFFECT declaration part is not fully implemented; EFFECT clauses are accepted and checked but not utilized in proofs.

9.1.2 Restrictions

- Side effect detection for functions is safe, but overly restrictive. If a function has side effects, essentially the only way it can be used is alone in an assignment statement. Procedures with side effects present no problems.
- The Verifier's knowledge about multiplication is weak. Nothing prevents the user from building new rules about the multiplication operator, but performance would be much better if the knowledge were built-in.
- The built-in knowledge about definedness of arrays is limited; arrays must be initialized in strictly increasing order of subscript. However, it is possible to prove more lemmas about **arraytrue!** to allow more general initialization if desired.
- The target machine against which the Verifier verifies is the Ford Electronic Engine Control IV, and the 16-bit, twos complement restrictions of that machine are enforced by the verifier.
- There is no compiler code generator pass compatible with the verifier at present, so there is no way to run Pascal-F programs containing verification statements. There is a Pascal-F compiler for the EEC IV, but it is not available for distribution outside Ford.

9.1.3 Known bugs

9.1.4 Pass 1 (Compiler pass)

- Some VALUE statements generate unexpected syntax errors.

- The compiler pass is not as solid as we would like; the intermediate code generated for some operations confuses the decompiler in pass 2, resulting in fatal internal errors.

9.1.5 Pass 2 (Semantic analysis)

- FORWARD declarations will cause pass 2 to become confused about block numbers and an internal check will abort the Verifier.
- There is a worry that the semantics of the FOR loop exit test may not exactly match the compilers for the case where the bounds are near to arithmetic overflow or underflow. The Verifier's semantics are conservative but may not be conservative enough.
- Records with only one field can create ambiguities as to whether a reference to a data item refers to a field or the entire record. This can result in pass 2 internal errors.

9.1.6 Pass 3 (Path tracing)

- The optimization of verification conditions will sometimes cause a useful term to be omitted from a hypothesis of a verification condition. The ommitted term will be from a proof goal, and will be a mention of a function whose arguments contain no variables needed in the proof at that point. A work-around for this is known.

9.1.7 Pass 4 (Simplifier)

- Rule handling is unreasonably slow in the presence of many rules applicable to the same expression.
- In at least one known case, the numeric portion of the prover fails to find a proof for a simple formula known to be true.

9.1.8 Rule Builder

- Lemmas about nonrecursive functions are not effectively used by the Boyer-Moore prover. This severly limits the proof power of the system with respect to the integers.
- The manual sections on hints are inadequate.

9.2 Reporting trouble

Problems with the system should be reported to the address given in the preface. All trouble reports should include copies of the files in the scratch directory, the source program, and the error messages printed. Before submitting the trouble report, the verification should be rerun with the **-d** keyletter on. This will rerun the verification with all debug output turned on.

9.3 Acknowledgements

Pascal-F was developed at the Ford Scientific Research Laboratories in Dearborn, Michigan, by Dr Edward Nelson. The Verifier is the work of Dr. Scott Johnson, John Nagle, Dr. John Privitera, and Dr. David Snyder, of Ford Aerospace and Communications Corporation. Dr. Derek Oppen consulted on the theorem prover modifications. The assistance of Dr. Robert Boyer and Dr. Jay Moore, of the University of Texas at Austin, has been very valuable, and we are indebted to Dr. Steven German, of Harvard University, for his formulation of the problem of checking for run-time errors. Finally, I would like to thank Dr. Shaun Devlin, of the Ford Motor Scientific Research Labs, for his faith and encouragement over the two years of the project.

John Nagle

9.4 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.*

HOARE74 *C.A.R. Hoare, Monitors: An Operating System Structuring Concept, Comm. ACM 17, pp. 549-557 (October, 1974)*

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.*