Abstract:

The design of correct and robust programs is investigated. It is argued that the notion of an exception is a valuable tool for structuring the specification, design, verification, and modification of such programs. The syntax and semantics of a language with procedures and exception handling are presented. A sound and complete deductive system is proposed for proving total correctness and robustness properties of programs written in this language. Simple examples which illustrate the language and proof rules are included. A more realistic application of the proposed programming and verification tools in fault-tolerant system development is described in a companion paper (3).

Correct and Robust Programs

By

F. Cristian

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 183
February, 1983

© 1982 University of Newcastle upon Tyne. Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
The design of correct and robust programs is investigated. It is argued that the notion of an exception is a valuable tool for structuring the specification, design, verification, and modification of such programs. The syntax and semantics of a language with procedures and exception handling are presented. A sound and complete deductive system is proposed for proving total correctness and robustness properties of programs written in this language. Simple examples which illustrate the language and proof rules are included. A more realistic application of the proposed programming and verification tools in fault-tolerant system development is described in a companion paper (3).
1. INTRODUCTION

The purpose of a program invocation is to transform an initial machine state into a final state such that some intended postcondition will hold. Let us call the set of initial states for which a program $P$ terminates by establishing a specified postcondition $Q$ the standard domain of $P$ with respect to $Q$ (from [4] one knows that the characteristic predicate of this domain is the weakest precondition $\text{wp}(P, Q)$). Classical program correctness theories tend to ignore what happens when a program is invoked outside its standard domain. In such circumstances the program either loops forever, or it terminates in a state which violates the specification $Q$, or it leads to a run-time error. Clearly such behavior is not desirable. In practice there is a need for robust programs which have a well-defined behavior for all possible inputs.

Let us call an invocation of a program outside its standard domain an exception occurrence. Exception handling is that part of programming which is concerned with the possibility of such occurrences instead of treating such a possibility as a problem. To achieve robustness, exception occurrences must be appropriately handled. This paper presents programming and verification tools suitable for developing programs which are both correct and robust.

The motivation for using an exception mechanism in order to structure the design of robust programs is discussed in §2.3. The syntax of a deterministic programming language with a simple exception mechanism is presented in §4. A semantic definition of this language is given in §5 in terms of predicate transformers. This semantics enables one to derive necessary and sufficient conditions for standard or exceptional program termination, but its direct use in program proofs may lead to laborious calculations. In order to structure such proofs a deductive system for proving sufficient conditions for correct standard and exceptional program behavior is presented in §6. This system is sound (only true facts can be
proved) and complete (every true fact can be proved). Simple examples which illustrate the language and proposed proof methods are included. A more realistic example may be found in a companion paper [3]. There, we specify, design, and prove the correctness of a system which is tolerant of certain classes of user and hardware faults (in this paper we assume that the hardware works properly).

2. EXCEPTION MECHANISM

In order to illustrate the problems encountered when handling exception occurrences let us give a simple example. Consider a procedure

\[
\text{proc } A(i\text{:int}, v\text{:int})
\]

with an integer value-result (vr) parameter \(i\) and an integer value (v) parameter \(j\), whose standard specification is \(i=(i_0+j_0)^2\) (\(i_0\) denotes the initial value of \(i\)). Suppose that in view of the goal to be achieved, the body of \(A\) is defined to be the sequential composition of the commands \(C_1, C_2\) below:

\[
\begin{align*}
(C1) & \quad i:=i+j ; \\
(C2) & \quad i:=i\cdot i .
\end{align*}
\]

Let \(\text{MI}\) be the set of machine representable integers and assume that the hardware interpreter checks for overflow on each integer operation. If one cannot guarantee that \(A\) is always invoked in a state which satisfies \(i+j \in \text{MI}\), an overflow exception occurrence may be detected when \(C_1\) is invoked. To insert an explicit run-time check before \(C_1\), in order to detect whether \(i+j \in \text{MI}\) holds, would be redundant with the overflow check which is performed by the lower level interpreter in any case. Now suppose that an overflow is detected when \(C_1\) is invoked. In such a case it does not make any sense to take the normal continuation of \(C_1\)
(that is, invoke the next command C2 following the ";" sequencer after C1) since this overflow detection reveals the impossibility of achieving the goal \(i=(i_0+j_0)^2\). Indeed, if \(i_0+j_0 \notin \text{MI}\) there exists no \(i \in \text{MI}\) such that \(i=(i_0+j_0)^2\).

This example illustrates a characteristic of exception occurrences: once such an event is actually detected it is no longer sensible to continue the standard execution of a program. Thus, in order to handle such events, it is necessary to allow for an occasional (exceptional) alteration of the standard sequential composition rule for operation invocations. An *exception mechanism* is a language control structure allowing one to express that the standard continuation of an operation invocation is to be replaced by an exceptional continuation when an exception occurrence is detected.

Before presenting in detail the syntax and semantics of the exception mechanism to be discussed in this paper, let us introduce its essential characteristics by referring to the simple example above.

An exceptional continuation for a command can be defined by using an *exception label*. For instance, the designer of the procedure A can declare an ow (overflow) exception label for A by writing

\[
\text{proc A}(\nu i:\text{int}, \nu j:\text{int})[\text{ow}].
\]

This declaration warns a user that A has two exit points: a standard one, which can be thought as being the ";" which follows an A invocation, and an exceptional one, which will be denoted "ow". The intention is that A should terminate at ";" when it can accomplish its standard goal \(i=(i_0+j_0)^2\) and that it should terminate at "ow" when it cannot. If A terminates at "ow" we say that A signals the occurrence of the ow exception.
In [4] the notion of a predicate transformer is proposed for specifying the semantics of one entry/one exit commands. When an exception mechanism is included in a language, most commands become one entry/multi exit. In order to specify the semantics of such commands we will generalize the notion of a predicate transformer introduced in [4].

We define the semantics of a one entry/multi exit command \( C \) to be a set of predicate transformers. As many predicate transformers as \( C \) has exit points. One of them will correspond to Dijkstra's predicate transformer, in that, for any postcondition \( Q \), it will give the weakest precondition under which \( C \) is guaranteed to terminate at ";" with \( Q \). We denote this standard predicate transformer by

\[
wp(C,;Q)
\]

instead of \( wp(C,Q) \) because we want to make explicit the fact that the interest is in termination at ";" (that is, normal termination). For each exceptional exit point "e" of \( C \), an exceptional predicate transformer

\[
wp(C,e,Q)
\]

will give, for any postcondition \( Q \), the weakest precondition for which \( C \) is guaranteed to terminate at "e" with \( Q \).

For example, the intended semantics of the procedure \( A \) can be specified by two predicate transformers. The first specifies that \( A \) terminates normally by assigning to its value-result argument \( m \) the value \((m+n)^2\) iff this value is machine representable:

\[
wp(A(m,n),;Q) \triangleq (m + n)^2 \epsilon \text{MI} \land Q[(m + n)^2/m]. \tag{1}
\]
The expression $Q[b/a]$ denotes the result of substituting every (free) occurrence of $a$ in $Q$ by $b$. The second predicate transformer specifies that $A$ signals $ow$ without changing its value-result argument whenever $(m+n)^2 \notin MI$ holds initially:

$$wp(A(m,n), ow, Q) \triangleq (m + n)^2 \notin MI & Q.$$  \hspace{1cm} (2)

If we denote "T" the constant true predicate, from (1) and (2) it follows that

$$wp(A(m,n), ;, T) \lor wp(A(m,n), ow, T) = T.$$ \hspace{1cm} (3)

That is, for any initial state of its arguments, $A$ terminates at a declared exit point (the ",;" exit point is considered implicitly declared for any command). In general, a program $P$ with $k \geq 0$ declared exceptional exit points $e_1, e_2, \ldots, e_k$ is robust if $P$ terminates at a declared exit point for any possible input state:

$$wp(P, ;, T) \lor \bigvee_{i=1}^{k} wp(P, e_i, T) = T.$$ \hspace{1cm} (R)

The robustness property defined above is different from that mentioned in [4], where a program $P$ is termed "robust" if it aborts whenever it is invoked in an initial state which does not satisfy the precondition specified for $P$.

The notion of correctness for one entry/multi exit programs is a straightforward extension of that for one entry/one exit programs. In [1] we proposed specifying the intended behavior of a program $P$ which signals $k \geq 0$ exceptions $e_1, \ldots, e_k$ by a set of pairs of pre/postconditions $(R_i, S_i)$, $i = 0, 1, \ldots, k$. One of the pairs $(R_0, S_0)$ corresponds to the classical pre/postconditions discussed in [4,5] and describes the intended standard behavior of $P$. The other pairs $(R_i, S_i)$, $i \geq 1$, specify the intended exceptional behavior of $P$. A program $P$ is correct with respect to such a specification if it terminates at ",;" with $S_0$ when it is invoked
in a state which satisfies $R_0$ and if it terminates at "$e_i$" with $S_i$ when it is invoked in a state which satisfies $R_i$, $i \geq 1$:

\[
(R_0 \Rightarrow \text{wp}(P, S_0)) \land \land_{i=1}^{k} (R_i \Rightarrow \text{wp}(P, e_i, S_i)) .
\]

(C)

Let us investigate now on our example what may happen after an exception occurrence is signalled. A user of the procedure $A$ can define the exceptional continuation, when ow is signalled, to be an *exception handler* $H$ by textually associating $H$ with the "ow" exit point of $A$:

\[
A(m,n)[\text{ow}:H] ;
\]

(4)

We term (4) a *protected command invocation*. By associating a handler with the "ow" exit point of $A$, the invoker is certain that after any $A$ invocation control returns either at ";;" or at "ow". It is the robustness property (3) of $A$ which excludes other outcomes. If, after a return at "ow", the handler $H$ terminates normally, (4) terminates normally. Thus, (4) terminates at ";;" either if $A$ does not signal ow (in which case $H$ is not invoked) or if $A$ signals ow and $H$ terminates normally:

\[
\text{wp}(A(m,n)[\text{ow}:H] ; ; Q) = \text{wp}(A(m,n) ; ; Q) \lor \text{wp}(A(m,n), \text{ow}, \text{wp}(H ; ; Q)) .
\]

(5)

There is, however, no requirement that a handler should always terminate normally. A handler can signal an exception $e$ if it terminates with a "$\triangleright e$" *exceptional sequencer*. A "$\triangleright e$" construct is allowed to occur only inside square brackets. One may understand such a sequencer (called "signal" in [1,7] and "raise" in [6]) as being a forward jump out of the syntactic unit in which the "$\triangleright e$" occurs. The target of the jump is the "$e$" exit point of that unit.
For example, in order to implement the behavior specified by (1,2) one may construct the procedure A as follows:

\[
\begin{align*}
\text{proc } A & (v: \text{int}, v: \text{int})[\text{ov}] ; \\
i & := i + j[\text{ov} : > \text{ow}] ; \\
i & := i * i[\text{ov} : i := i - j > \text{ow}] .
\end{align*}
\]

**Figure 1.** A (simple) robust procedure with exceptions

If the overflow (ov) language defined exception is signalled by C1, the 'next' command C2 is not invoked. Instead C1 terminates at its "ov" exit point, where the ">ow" sequencer directs control to the "ow" exit point of A. If C1 terminates normally but C2 signals ov, the value of i is reset to \(i_0\) before the "ow" exit of A is taken. If both C1 and C2 terminate normally, the termination of A is normal. Let us describe in more detail the meaning of the constructs used in A.

If a ">e" command which occurs in a protected command is invoked, control will not reach the ";" which follows that protected command

\[
wp(>e;.,Q) \triangleq F \tag{6}
\]

(\(F\) denotes the constant false predicate). The occurrence of a ">e" sequencer in a protected command defines the existence of an "e" exit point for that command. When the ">e" sequencer is invoked control goes to that "e" exit point

\[
wp(>e,e,Q) \triangleq Q . \tag{7}
\]

Like the ";" standard sequencer, the ">e" exceptional sequencer has no effect on the state of program variables.
For example, from (5,6) one can infer the general theorem that the standard meaning of any protected command \( C[e;C_1;\triangleright f] \) is the same as that of the nonprotected \( C \):

\[
wp(C[e;C_1;\triangleright f],;Q) = wp(C,;Q) .
\]  

This is an important property of the exception mechanism: in order to derive the standard semantics of a program in which all the handlers terminate with "\( \triangleright \)" sequencers, one needs not be concerned with the exceptional program text between "[" and "]" separators. For example, the standard meaning of the body \( B(A) \) of the procedure \( A \) can be derived after deleting all the text surrounded by square brackets:

\[
wp(B(A),;Q) = wp(SB(A),;Q) = (i + j)^2 e M I \& \lfloor (i + j)^2 / i \rfloor
\]

where \( SB(A) \equiv i := i + j; \ i := i * i \) denotes the standard code of \( A \). When computing (9) we have used the fact that for a sequence of commands to terminate normally it is necessary (and sufficient) that the component commands terminate normally:

\[
wp(C_1;C_2,;Q) \triangleq wp(C_1,;wp(C_2,;Q)) .
\]

When a command in a sequence signals an exception \( e \) the next commands in the sequence are not invoked and the whole sequence terminates at "e":

\[
wp(C_1;C_2,e,Q) \triangleq wp(C_1,e,Q) \lor wp(C_1,;wp(C_2,e,Q)) .
\]

That is, \( C_1;C_2 \) can signal \( e \) either if \( C_1 \) signals \( e \) (in which case \( C_2 \) is not invoked) or if \( C_1 \) terminates normally but \( C_2 \) signals \( e \). The semantic clause (11) reflects the main change brought by an exception mechanism into a language: the possibility of escaping from standard sequencing when exceptions occur. By using (7,11) one can derive the exceptional predicate transformer of the body \( B(A) \) of \( A \) as being

\[
wp(B(A),ow,Q) = (i + j)^2 \notin M I \& Q .
\]
The actual meaning (9,12) of A resembles its intended meaning (1,2). As will be shown later, from (9,12) one can infer that any A invocation behaves as specified by (1,2).

This terminates the introductory presentation of the three keywords "[", "]" and "\triangleright\" of the exception mechanism that we propose. At this point, however, a skeptical reader may ask whether the inclusion of such a control structure in a language is really useful.

3. A STRUCTURING TOOL

The design of exception mechanisms is one of many programming language issues which is a subject of current debate. Some computer scientists consider an exception mechanism as being an unnecessary "feature". Others would argue that it is better to avoid exceptions than to handle them. It is not difficult, for example, to write a "structured" version SA of our procedure A which uses only one entry/one exit commands and avoids overflow occurrences by explicitly checking boundary conditions:

```
proc SA(vr i:int, v j:int, r done:int);
if (0≤j) and (i≤max-j) or (j<0) and (min-j≤i)
then i:=i+j;
  if (0<i) and (i≤max div i) or (i<0) and (-i≤max div-i) or (i=0)
  then i:=i*i;
  done:=1
else i:=i-j;
  done:=0
fi
else done:=0
fi.
```

**Figure 2.** A (simple) robust procedure without exceptions

The *and, or* language defined Boolean operations are assumed to be noncommutative, as otherwise, the SA procedure would not be robust. For SA one does not need two predicate
transformers to describe its effect; in contrast to A, one is sufficient:

\[
wp(SA(m,n,p),\ell,Q) = (m + n)^2 e MI & Q[(m + n)^2/m, 1/p] \lor (m + n)^2 \not\in MI & Q[0/p].
\] (13)

Protected constructs like (4) can be programmed by explicitly checking whether the value of an actual return code \( p \) is 0 or 1, so as to direct control to the right 'next' statement in the program which invokes \( SA \) (and thus simulate the notion of an exceptional continuation discussed in §2):

\[
SA(m,n,p);
\text{ if } p=0 \text{ then } H \text{ else } "\text{standard continuation}" i.
\]

Why, then, should one investigate exceptions and worry about exception mechanisms?

It is, of course, impossible to base one's arguments on a single example such as the simple example considered above. What will be said in the remainder of this section is the result of our practical experience in designing (and sometimes verifying the correctness of) a reasonably large number of nontrivial robust programs. Most of them were initially written in a 'structured' fashion by using only one entry/one exit commands and then rewritten by using the exception mechanism sketched in §2.

It seems clear that in order to write trustworthy software, one should strive for simplicity [4,5]. Therefore, it is important to see whether the contribution of any new language construct to simplify typical programming tasks outweighs the additional complexity introduced by the construct in the language. The claim is made that the tasks of specifying, constructing, verifying and modifying robust programs can be made simpler if a simple exception mechanism, like that outlined in §2, is integrated into a language.

By decomposing the input space of a program into disjoint standard and exceptional domains, one can specify and understand what the program does on each of these domains
separately. The exceptional domain in its turn may be divided further into smaller pieces (see, for example, §7) and the behavior on each of these subdomains can also be analyzed separately. Since the functions computed on distinct domains are in general different, this allows one to concentrate on one thing at a time. Such a "divide and conquer" approach is of great value, especially when dealing with larger problems (see, for example, [1,3]), but even on our small example, one may remark that it is easier to comprehend the specifications (1,2) separately and then combine them (3) in order to obtain a global view, than to comprehend (13) at once.

The exception mechanism introduced in §2 allows a clear textual separation of the program commands executed when no exceptions occur from those executed when exceptions occur. Because it allows for direct expression of the alterations which have to be made to standard sequencing when exceptions occur, there is no need to use auxiliary program variables (like "done" and "p") or cascades of conditionals in order to direct control to the right 'next' commands. In general this leads to shorter programs and, it is believed, also improves program readability, once the "[", "]", "≫" new symbols have been understood. Programs may be developed incrementally by first concentrating on their standard behavior and then by taking into account the handling of exceptional situations.

The task of verifying the correctness of robust programs can be factored into simpler tasks: the correctness proof of a program which may signal $k \geq 0$ exceptions consists of $k+1$ independent subproofs: one standard correctness proof and $k$ exceptional correctness proofs (e.g., in §7 and [1,3]). Since no auxiliary variables need be used to direct control to the appropriate continuations, the formulae which have to be manipulated in such subproofs tend also to be shorter than those which would have to be manipulated if only one entry/one exit commands were used.
Program modifiability is also improved. For instance, it is reasonable to assume that library programs are built with some attempt at robustness, as one does not know who will use them. Now, if one can guarantee that any invocation of a library program from another (thoroughly verified) program will not lead to exception occurrences, an optimized version may be easily produced by just deleting the text between square brackets. For example, if one can prove that the procedure of Figure 1 is always invoked in its standard domain \((i+j)^2 \epsilon M1\), the task of producing an optimized version by removing the exceptional text is straightforward. In order to achieve a similar effect for the procedure of Figure 2, much more analysis of the program code would be required.

One may remark that the arguments above, forwarded to substantiate the claim that exceptions are a useful software structuring tool, relate to the principle of "separations of concerns" [4] well known in programming methodology. The motivation behind the work on exception handling is to make the practical application of this principle easier.

4. SYNTAX

In the language that we want to present, a program will consist of a name \(N\), a set of declarations and a command \(C\):

\[
\text{PROG ::= } \text{prog } N \ [\text{EL}] \ \text{SDECL} ; \ C .
\]

The declarations state a list of exceptional exit points \(e\) at which \(N\) may terminate

\[
\text{EL ::= e } | \ e, \text{EL} .
\]

as well as the variables \(v\) and the procedures \(Pr\) which are known in \(N\):

\[
\text{SDECL ::= DECL } | \ \text{DECL} ; \ \text{SDECL} .
\]
DECL ::= \mid \textit{var} \textit{v:} \text{int} \mid \textit{proc} \text{Pr}(v \textit{vp, vr} vrp, r \textit{rp})[EL] \text{SDECL; C} .

For simplicity we consider only program variables of type (machine representable) integer and procedures with one value parameter (vp), one value-result parameter (vrp) and one result parameter (rp). More complex (machine representable) data types are discussed in [1]. The semantics of procedures which either do not have all the above kinds of parameters or have several parameters of the same kind should be derivable from the semantic definitions in §5. We assume that each variable which occurs in a procedure body is either locally declared or is a parameter, and that procedures are not recursive.

Commands C may be empty, sequencers, simple (S), protected (PC), or may be composed sequentially from other commands:

\[ C ::= \mid \text{De} \mid S \mid \text{PC} \mid C ; C . \]

A simple command S is either an assignment, a conditional, a loop, a block, or a procedure call:

\[ S ::= v := \text{ie} \mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \mid \text{while } B \text{ do } C \text{ od} \mid \text{begin } C \text{ end} \mid \text{Pr}(va,vra,ra) . \]

Assignment commands may signal the language defined exceptions overflow (ov) and division by zero (dz). We do not give any detailed syntax for Boolean expressions B and assume that their evaluation does not lead to overflow or division by zero exceptions. There is no difficulty in defining a semantics which takes into account such possibilities. We refrain from doing so in order to keep our presentation as simple as possible. We assume that the arguments va, vra, ra of any procedure call are declared variables, all distinct and different form the parameters vp, vrp, rp.
There are two kinds of protected constructs PC:

\[ PC ::= [B:C] \mid S[HAL] \].

The first kind makes the invocation of a handler C dependent upon the truth of a Boolean expression B. The second kind associates one or several handlers with a set of exceptional exit points of a simple command:

\[ HAL ::= EL:C \mid EL:C, HAL \].

The following context sensitive constraints must (and can) be enforced by a compiler: all the exception labels which occur in a handler association list HAL must be distinct, and \( \triangleright \) exceptional sequencers can occur only in protected constructs (inside square brackets) in order to terminate handlers. As pointed out in [1] the last constraint is fundamental for obtaining a clear separation between standard and exceptional program semantics.

An integer expression "ie" is either a program constant \( m \), a program variable \( v \), or the result of applying a binary language operation "\( \circ \)" to integer expressions:

\[ ie ::= m \mid v \mid (ie) \mid ie \circ ie \].

We consider addition, subtraction, multiplication and integer division and we assume that parenthesis are used whenever necessary to avoid ambiguous parsing of expressions:

\[ \circ ::= + \mid - \mid \ast \mid div \].

We terminate our discussion about syntax by an example of a syntactically well-formed program fragment:
\textit{proc FACT}(v n:int, r: int)[neg, ow]\]
\begin{verbatim}
var k:int;
[n<0: $\triangleright$ neg];
k:=0;
while k<n 
do k:=k+1;
    r:=r*k[ov: $\triangleright$ ow]
od.
\end{verbatim}

Figure 3. A robust procedure which computes factorials

When invoked with \( n>0 \) and \( n! \in \mathbb{N} \), this procedure terminates normally with \( r=n! \); otherwise one of the exceptions negative (neg) or overflow (ow) is signalled. Note that, unlike most of the procedures given in the literature for computing factorials, we have not ignored the possibility of arithmetic overflow. (On a 16-bit machine, for example, an overflow will occur whenever one attempts to compute the factorial of an integer which is greater than 6.)

5. PREDICATE TRANSFORMER SEMANTICS

A denotational style is adopted for specifying the semantics of our language: the meaning of each syntactic construct is specified only in terms of the semantics of its components. The weakest precondition semantic function \( \text{wp} \), that we intend to define, is of type:

\[ \text{wp} : \text{COMMAND} \rightarrow \text{EXITPOINT} \rightarrow \text{PREDICATE} \rightarrow \text{PREDICATE}. \]

For each \( C \in \text{COMMAND}, x \in \text{EXITPOINT}, \) and postcondition \( Q \in \text{PREDICATE} \), \( \text{wp}(C,x,Q) \) yields the weakest precondition which guarantees that \( C \) will terminate at \( x \) in a state in which \( Q \) will be true.

In order to express predicates we use propositional formulae over the integer structure

\[ \mathcal{P} = \langle \mathbb{Z}, +, -, \times, \text{div}, <, =, \min, 0, 1, \max \rangle. \]
The +, −, * operators over the (infinite) set of integers $\mathbb{Z}$ are total, and so are the < and = relational operators. The div operator is partial: it is undefined when its second operand is zero. The min, max constants correspond to the smallest and greatest machine representable integers, respectively. (Note that the same graphic symbols are used to denote both the operators on $\mathbb{Z}$ and the operations available in our programming language. Which meaning is intended should be clear from the context.)

Let $v$ stand for a program variable, $m$ stand for a program constant and $z$ stand for any integer in $\mathbb{Z}$. The language of terms $t$ over $\mathcal{P}$ is defined by the rule

$$t ::= v \mid m \mid z \mid (t) \mid t \circ t \mid \text{if } P \text{ then } t \text{ else } t$$

$$\circ ::= + \mid - \mid * \mid \text{div}.$$ 

The language of propositional formulae $P$ over the structure $\mathcal{P}$ is generated by the rule

$$P ::= T \mid F \mid t=t \mid t<t \mid (t) \mid \text{if } P \text{ then } P \text{ else } P.$$ 

A propositional formula $P$ with variable symbols $v_1, v_2, ..., v_n$ will be called well-defined if for any constants $m_1, m_2, ..., m_n$ the constant formula $P[m_1/v_1, ..., m_n/v_n]$ is either $T$ or $F$ (that is, is not undefined). For example, "$x \text{ div } y < x"$ is not well-defined but "$\text{if } 0 < y \text{ then } x \text{ div } y < x \text{ else } T"$ is.

The assertion language $\mathcal{L}$ which will be used to specify the semantics of syntactically well-formed programs will be a subset of the language of propositional formulae over the structure $\mathcal{P}$, namely the set of well-defined propositional formulae over $\mathcal{P}$. We use the syntactic variables $P, Q, R, B$ to range over assertions in $\mathcal{L}$. In writing such formulae we use a set of standard abbreviations like "$\neg P$" for "$\text{if } P \text{ then } F \text{ else } T$", "$P \& Q$" for "$\text{if } P \text{ then } Q \text{ else } F$", "$P \Rightarrow Q$" for "$\text{if } P \text{ then } Q \text{ else } T$" and "$P \lor Q$" for "$\text{if } P \text{ then } T \text{ else } Q$". If $I$ is a
finite index set and $P_i, i \in I$, are assertions we abbreviate the finite conjunction or disjunction of these assertions by $\bigwedge_{i \in I} P_i$ and $\bigvee_{i \in I} P_i$, respectively. We also use other obvious abbreviations like $i \neq j$ for $\neg (i=j)$, $i \leq j$ for $\neg (i < j)$, $i \in M$ for $(\min \leq i \land i \leq \max)$, $e \in \{e_1, e_2, ..., e_n\}$ for $\bigvee_{i=0}^{n} (e = e_i)$, etc.

Although the assertion language $\mathcal{L}$ is very simple, compared to the first order languages usually employed when writing assertions about programs, $\mathcal{L}$ is nevertheless expressive enough for specifying the semantics of any program in our language. This is a consequence of the fact that the integers manipulated by programs are finite. We assume that any program variable $v$ has at any time some value in the range $[\min, \max]$:

$$\text{if } v \text{ is a program variable then } (\min \leq v) \land (v \leq \max). \quad (f1)$$

This corresponds to the way integer variables are represented in storage by fixed length sequences of bits, where any possible bit configuration represents some element of MI. Similarly we assume that any program constant $m$ denotes some machine representable integer:

$$\text{if } m \text{ is a program constant then } (\min \leq m) \land (m \leq \max). \quad (f2)$$

Since the set of variables of any program is finite, and any program variable can only be in a finite number of distinct states, it follows that any program has only a finite set $S$ of distinct states. (We denote the cardinality of $S$ by $|S|$.) Thus, in order to express subsets of $S$ (or equivalently predicates in $S \to \{\text{tt, ff}\}$) there is no need to use quantifiers: propositional logic is sufficient. Since to any assertion $P \in \mathcal{L}$ corresponds naturally a predicate in $S \to \{\text{tt, ff}\}$ and any such predicate is expressible by an assertion in $\mathcal{L}$, we need not distinguish in what follows between assertions and predicates.
The semantics of programs will be defined by induction on their syntactic structure. The empty command (called "skip" in [4]) always terminates normally without changing the program state:

\[ \text{wp}(x, Q) \triangleq \text{if } x = ; \text{ then } Q \text{ else } F. \]  \hspace{1cm} (S1)

While the empty command always passes control to the 'next' command which follows after ";", an "e" sequencer never passes control to this 'next' command. When a "e" sequencer is invoked control is passed to the "e" exit point of the smallest enclosing command:

\[ \text{wp}(e, x, Q) \triangleq \text{if } x = e \text{ then } Q \text{ else } F. \]  \hspace{1cm} (S2)

In order to define the semantics of integer assignments, it is first necessary to define the conditions under which the evaluation of an integer expression "ie" leads to a machine representable result (rep), to an overflow (over), or to an undefined result (undef). These three conditions are defined by induction on the syntactic structure of integer expressions:

\[ \text{rep}(m) \triangleq T, \text{ rep}(v) \triangleq T, \text{ rep}((ie)) \triangleq \text{rep}(ie), \]

\[ \text{rep}(ie_1 \circ ie_2) \triangleq \text{rep}(ie_1) \& \text{rep}(ie_2) \& (\text{if } \circ = \text{div then } ie_2 \neq 0 \text{ else } ie_1 \circ ie_2 \in \text{MI}) \]

\[ \text{over}(m) \triangleq F, \text{ over}(v) \triangleq F, \text{ over}((ie)) \triangleq \text{over}(ie), \]

\[ \text{over}(ie_1 \circ ie_2) \triangleq \text{over}(ie_1) \vee \text{rep} (ie_1) \& \text{over}(ie_2) \vee (\circ \neq \text{div}) \& \text{rep}(ie_1) \& \text{rep}(ie_2) \& (ie_1 \circ ie_2 \notin \text{MI}) \]

\[ \text{undef}(m) \triangleq F, \text{ undef}(v) \triangleq F, \text{ undef}((ie)) \triangleq \text{undef}(ie), \]

\[ \text{undef}(ie_1 \circ ie_2) \triangleq \text{undef}(ie_1) \vee \text{rep}(ie_1) \& \text{undef}(ie_2) \vee (\circ = \text{div}) \& \text{rep}(ie_1) \& \text{rep}(ie_2) \& (ie_2 = 0). \]

The semantics of an assignment is:
\[
wp(v = ie,x,Q) \triangleq \begin{cases} 
  \text{if } x = \text{ then } \text{rep(ie)} & \& Q[ie/v] \text{ else} \\
  \text{if } x = ov \text{ then } \text{over(ie)} & \& Q \text{ else} \\
  \text{if } x = dz \text{ then } \text{undef(ie)} & \& Q \text{ else } F
\end{cases}
\]  

(S3)

Since the predicates "rep", "over", and "undef" have the property that, for any syntactically legal expression evaluation, exactly one of them is T and the others are F, assignments are robust and deterministic. The noncommutative "&" in (S3) ensures that only well-defined terms are substituted in conditions Q, so when Q is well defined the result of such substitutions will also be well defined. In specifying the exceptional semantics of assignments, we have chosen to leave the value of the left-hand side variable v unchanged whenever one of the ov or dz predefined exceptions is signaled.

A conditional command inherits the exit points of its alternatives: it terminates at an exit point x with a postcondition Q if the alternative which is taken terminates at x with Q:

\[
wp(if \ B \ then \ C1 \ else \ C2 \ fi,x,Q) \triangleq B \& wp(C1,x,Q) \lor \neg B \& wp(C2,x,Q) .
\]  

(S4)

Consider now a loop command \( L \triangleq while \ B \ do \ C \ od \). The loop L inherits all the exit points of its body C. The condition for L to terminate at ";" after zero iterations with a postcondition Q is the truth of

\[
S_0 \triangleq \neg B \& Q
\]

If, when L is invoked, B is true, the body C is executed. If C terminates at ";" in a state satisfying \( S_0 \), the loop terminates normally:

\[
S_1 \triangleq B \& wp(C,; ,S_0)
\]

If C terminates by signalling an exception e, the whole loop terminates by signalling e:

\[
X_1 \triangleq B \& wp(C,e,Q)
\]
In general, in order to see whether $L$ terminates after $i$ iterations (either at "," or at some "e") one has to evaluate the conditions $S_i$, $X_i$ defined recursively below:

$$S_i \triangleq B \land \text{wp}(C;,S_{i-1})$$

$$X_i \triangleq B \land \text{wp}(C;,X_{i-1}) .$$

How many such conditions needs one to evaluate in order to convince himself whether a loop terminates with some desired postcondition? In order to give an answer we need the following lemma:

**Lemma:** $S_i \land S_j = F$ and $X_i \land X_j = F$ for any positive integers $i \neq j$ and any exceptional exit point e.

This lemma expresses the fact that the sets of program states which satisfy the $S_i$, $X_i$ conditions defined above are pair-wise disjoint: there exists no state from which a loop can terminate both after exactly $i$ iterations and after exactly $j$ iterations (unless $i$ and $j$ are equal). The proof (which relies on the fact that only deterministic commands can be used in the loop body) is omitted.

**Theorem I:** If a loop $L$ does not terminate in less than $|S|$ steps, it never terminates:

$$z \geq |S| \implies S_z = F,$$

$$z > |S| \implies X_z = F .$$

We prove only the standard case (the exceptional case is similar). Assume that there exists a $z \geq |S|$ for which $S_z$ is not the constantly false predicate. Then there exists a state $s_0$ which satisfies $S_z$. By the definition of the $S_i$ predicates, $i \geq 0$, this means that, if the loop $L$ is
invoked in \( s_0 \), it will pass through a sequence of intermediate states \( s_1, \ldots, s_{z-1} \), such that for each \( s_i, 0 \leq i \leq z-1 \) the predicate \( S_{z-i} \) is true, and will terminate in a state \( s_z \) in which \( S_0 \) will be true. Because of our previous lemma, all the intermediate states \( s_0, s_1, \ldots, s_z \) are distinct. So we have constructed a sequence of \( z+1 > |S| \) distinct program states, which contradicts the fact that there exist only \( |S| \) distinct program states. Thus, for \( z \geq |S| \) the \( S_z \) conditions are constantly false.

We are now in a position to answer our initial question: In order to see whether a loop \( L \) terminates (either normally or exceptionally) one has to evaluate at most \( |S| \) successive \( S_i, X_i \) conditions. If all of them are false the loop is divergent. Thus, the predicate transformers of a loop \( L \) are the propositional formulae:

\[
wp(\text{while } B \text{ do } C \text{ od}, x, Q) \triangleq \begin{cases} \text{if } x = ; & \text{then } \bigvee_{i=0}^{\lfloor S \rfloor - 1} S_i \text{ else } \bigvee_{i=1}^{\lceil S \rceil} X_i. \\ \end{cases} \tag{S5}
\]

For example, by using (S5) one can show that, for any initial value \( v_0 \in MI \), the loop

\[
\text{while true do } v := v + 1 \text{ od}
\]

terminates by signalling an overflow exception. Indeed, for \( Q = T \), \( X_1 = (v = \text{max}) \), \( X_2 = (v = \text{max}-1) \), ... and an inductive argument shows that \( X_i = (v = \text{max}-i+1) \). Thus, for \( i = \text{max}-v_0+1 \), \( X_i \) (and hence the disjunction of \( X_i \)'s) is true.

In more complex cases finding the necessary and sufficient conditions for standard or exceptional loop termination may lead to laborious calculations. The usual method for coping with the complexity inherent in loops is to understand them in terms of an invariant condition \( I \) and a nonnegative (monotonically decreasing) variant integer expression \( U \) [4]. The invariant describes what remains unchanged during a loop execution. The variant expression \( U \) yields, for given values of program variables, an upper bound on the number of
iterations still to be performed. The values that U may take need not be machine representable (U is a proof theoretical concept which needs not be evaluated at run time).

The following theorems are useful in showing that the truth of some condition I is sufficient to guarantee that a loop will terminate with some desired postcondition Q.

**Theorem 2:** If \(I \Rightarrow (0 \leq U), \ I \& (U = 0) \Rightarrow \neg B, \ I \& \neg B \Rightarrow Q,\) 
\[I \& B \& (U \leq t) \Rightarrow \text{wp}(C, ; I \& (U < t))\]

*then* \(I \Rightarrow \text{wp} \ (\text{while} \ B \ \text{do} \ C \ \text{od};, \ Q)\)

**Theorem 3:** If \(I \Rightarrow (0 \leq U), \ I \Rightarrow B, \ I \& (U = 0) \Rightarrow \text{wp}(C, e, Q),\) 
\[I \& (0 < U) \& (U \leq t) \Rightarrow \text{wp}(C, ; I \& (U < t))\]

*then* \(I \Rightarrow \text{wp} \ (\text{while} \ B \ \text{do} \ C \ \text{od};, \ e, \ Q)\)

The above theorems can be proved by first showing (by induction on \(t \geq 0\)) that the hypotheses of Theorem 2 imply

\[I \ & \ (U \leq t) \Rightarrow \bigvee_{i=0}^{t} S_i\]

and, the hypotheses of Theorem 3 imply

\[I \ & \ (U \leq t) \Rightarrow \bigvee_{i=1}^{t+1} X_i\]

The conclusions then follow from the above lemmas, Theorem 1 and Definition (S5).

Before giving the semantics of the remaining simple commands (blocks and procedure calls) let us recall the semantics of a sequence of commands introduced in §2. The formulae (9,10) given there can be rewritten as:

\[\text{wp}(C_1; C_2, x, Q) \triangleq (if \ x \neq; \ then \ \text{wp}(C_1, x, Q) \ else \ F) \ \lor \ \text{wp}(C_1, ;, \text{wp}(C_2, x, Q))\] . (S6)
In order to associate a unique handler with an exception label which may be signalled by several commands of a sequence, it is sufficient to enclose that sequence in a block and attach the handler to the block. A block inherits all the exit points of its body

\[
\text{wp}(\text{begin } C \text{ end}, x, Q) \triangleq \text{wp}(C, x, Q) \quad (S7)
\]

and is a simple command (according to the syntax in §4 handlers can be associated only with exceptional exit points of simple commands). Because of (S6, S7), an "e" exceptional exit point of a block 'joins' all the "e" exit points of the commands which occur in that block: the "e" exit is taken if any of these commands signals e. For example, the handler H below can be reached if either the first or the second commands of the block signal dz:

\[
\begin{align*}
\text{begin} & \quad i := i \div j \; ; \\
& \quad l := l \div k \\
\text{end} \; [dz:H].
\end{align*}
\]

Even if no handler was associated with the dz exit point of the above block, that exit point would still exist: it would be taken whenever either j or k are zero. The existence of a free exit point "e" for a command (no handler bound to it at the level of that command) implies the existence of another exit point "e" for the next enclosing command. If no handler is associated with the latter exit point, there will be another "e" exit point for the next enclosing command, and so on. In general, the target of a ">e" sequencer is the "e" exit point of the nearest enclosing simple command which has a handler associated with e. If such a handler exists, the program execution continues with it, otherwise, the entire program terminates at "e" (one may understand this as being termination with the error message e).

The ability to exit several nested syntactic constructs when an exception is detected helps to write shorter programs, but it should be used with great caution. Indeed, it is easy to lose track of some of the exceptional exit points of the inner blocks when too many blocks
are nested. When accompanied by a systematic robustness verification such negligence will manifest itself in the impossibility of proving the robustness of the outermost command of a program. But when no robustness verification is attempted such negligence may lead to the run-time situation where there exists no programmer defined handler for some exception which is signalled. An automatic association by a compiler or operating system of default handlers with unhandled exceptions can provide some degree of fault-tolerance but reliance on such techniques is risky [2]. In our opinion multi-level exits can be used safely only when accompanied by robustness proofs.

Consider now a procedure Pr declared as

$$
\text{proc } Pr(v \ v p, \ vr \ vrp, \ r \ rp) \ [E L] \ S D E C L ; \ C
$$

and assume that Pr invocations satisfy the syntactic constraints of §4. The semantics of a procedure call is:

$$
wp(Pr(va, vra, ra), x, Q) \overset{\Delta}{=} wp(init,; ,wp(C,x,wp(fin,; ,Q))) \tag{S8}
$$

where "init" denotes the initialization action "vp:=va; vrp:=vra" and "fin" denotes the finalization action "vra:=vrp; ra:=rp". For example, by using (S8) successively with x=; and x=ow, one can establish that the specifications (1,2) of §2 are correctly implemented by the procedure of Figure 1, whose body has the actual semantics (8,12).

A protected construct [B:C] terminates normally if either B is false (in which case the handler C is not invoked) or if B is true and C terminates normally. The construct can signal an exception e if B is true and C signals e:

$$
wp([B:C], x, Q) \overset{\Delta}{=} (if \ x = ; \ then \ \neg B & Q \ else \ F) \lor B & wp(C, x, Q) . \tag{S9}
$$

A protected construct of the second kind (see §4) has the general form
S[EL_1;C_1,...,EL_n;C_n].

It terminates normally if either S terminates normally (in which case none of the handlers C_i are invoked) or if S signals an exception e in some list EL_i and the handler C_i associated with this list terminates normally:

\[
wp(S[EL_1;C_1,...,EL_n;C_n],Q) \triangleq wp(S;,Q) \lor \bigvee_{i=1}^{n} wp(S,EL_i,wp(C_i;,Q)) \quad (S10')
\]

where \( wp(S,EL,Q) \) is an abbreviation for \( \bigvee_{e \in EL} wp(S,e,Q) \).

The protected command can signal an exception e either if S signals e and e is not handled by any of the handlers C_i (that is, if \( e \notin \bigcup_{i=1}^{n} EL_i \)) or if S signals an exception label f which is in a list EL_i and the handler C_i associated with this list terminates by signaling e:

\[
wp(S[EL_1;C_1,...,EL_n;C_n],e,Q) \triangleq \quad (S10'')
\]

\[
if e \notin \bigcup_{i=1}^{n} EL_i \ then \ wp(S,e,Q) \ else \ \bigvee_{i=1}^{n} wp(S,EL_i,wp(C_i,e,Q)).
\]

Since the "," standard exit point cannot occur in a list EL_i of exceptional exit points, the (S10', S10'') semantic definitions can be combined into the following general semantic clause:

\[
wp(S[EL_1;C_1,...,EL_n;C_n],x,Q) \triangleq \quad (S10)
\]

\[
\left( if x \notin \bigcup_{i=1}^{n} EL_i \ then \ wp(S,x,Q) \ else \ F \right) \lor \bigvee_{i=1}^{n} wp(S,EL_i,wp(C_i,x,Q)).
\]

The predicate transformer semantic function \( wp \), defined inductively by the clauses (S1-S10), satisfies a set of "healthiness" conditions similar to those discussed in [4]. We give them (without proof) below:

(H1) If \( Q \in \mathcal{L}, C \in \text{COMMAND}, x \in \text{EXITPOINT} \) then \( wp(C,x,Q) \in \mathcal{L} \)

(H2) \( wp(C,x,F) = F \)
(H3) if \( P \Rightarrow Q \) then \( \text{wp}(C,x,P) \Rightarrow \text{wp}(C,x,Q) \)

(H4) \( \text{wp}(C,x,P \& Q) = \text{wp}(C,x,P) \& \text{wp}(C,x,Q) \)

(H5) \( \text{wp}(C,x,P \lor Q) = \text{wp}(C,x,P) \lor \text{wp}(C,x,Q) \)

(H6) A command invocation may terminate at most at one exit point:
\[
x_1 \neq x_2 \Rightarrow \text{wp}(C,x_1,T) \& \text{wp}(C,x_2,T) = F
\]

(H7) If a command \( C \) does not contain a loop, it terminates at some exit point \( x \):
\[
C \text{ loop free} \Rightarrow \bigvee_{x \in \text{EXITPOINT}} \text{wp}(C,x,T) = T.
\]

6. DEDUCTIVE SYSTEM

Although predicate transformers provide a convenient way of specifying program semantics, their direct use in program proofs may lead to laborious calculations, mainly because of the loop semantics. Most often one is required to establish sufficient conditions for correct program behavior (the \( \text{wp} \) transformer gives necessary and sufficient conditions, which are usually neither needed nor simple). In order to structure program correctness proofs, the use of a deductive system, as suggested in [5], is more convenient. In this section we present such a deductive system for proving (total) correctness and robustness properties of programs with exceptions.

We take the set of all valid assertions in \( \mathcal{P} \) together with the \((f1,f2)\) postulates to be axioms. A set of general theorems about the predicate transformer semantics of \( \S \) are the proof rules of the system. In writing such theorems we use the notation \( P \{C\}x: Q \) to stand for \( P \Rightarrow \text{wp}(C,x,Q) \). If \( x=; \) then we adopt the convention of writing \( P \{C\} Q \) instead of \( P \{C\};: Q \). The interpretation of a statement \( P \{C\}x: Q \) in our system is "if \( C \) is invoked in a state in which \( P \) is true, \( C \) terminates at \( x \) in a state in which \( Q \) is true".
The following consequence rules are simple rewritings in terms of the new notation of properties (H3, H4, H5) of §5:

\[
\begin{align*}
P \Rightarrow S, \ S \{C\}x: R, \ R \Rightarrow Q & \\
P \{C\}x: Q
\end{align*}
\]  \hspace{1cm} \text{(R1)}

\[
\begin{align*}
P \{C\}x: Q, \ P \{C\}x: R & \\
P \{c\}x: (Q \& R)
\end{align*}
\]  \hspace{1cm} \text{(R2)}

\[
\begin{align*}
P \{C\}x: R, \ Q \{C\}x: R & \\
(P \lor Q) \{C\}x: R
\end{align*}
\]  \hspace{1cm} \text{(R3)}

The proof rules for assignment commands are

\[
\begin{align*}
P \Rightarrow \text{rep}(ie) \& Q[ie/v] & \\
P \{v := ie\} Q
\end{align*}
\]  \hspace{1cm} \text{(R4)}

\[
\begin{align*}
P \Rightarrow \text{over}(ie) & \\
P \{v := ie\} ov: P
\end{align*}
\]  \hspace{1cm} \text{(R5)}

\[
\begin{align*}
P \Rightarrow \text{undef}(ie) & \\
P \{v := ie\} dz: P
\end{align*}
\]  \hspace{1cm} \text{(R6)}

The rep, over, undef conditions are those defined in §5. The proof rule for conditional commands is

\[
\begin{align*}
P \& B \{C1\}x: Q, \ P \& \neg B \{C2\}x: Q & \\
P \{if \ B \ then \ C1 \ else \ C2 \ fi\}x: Q
\end{align*}
\]  \hspace{1cm} \text{(R7)}

The proof rules for loops are transcriptions in the new notation of Theorems 2,3 of §5:

\[
\begin{align*}
P \Rightarrow (0 \leq U), \ P \& (U = 0) \Rightarrow \neg B, \ P \& \neg B \Rightarrow Q, \\
P \& B \& (U \leq t) \{C\} P \& (U < t)
\end{align*}
\]  \hspace{1cm} \text{(R8)}

\[
P \{\text{while} \ B \ \text{do} \ C \ \text{od}\} Q
\]
\[ P \Rightarrow (0 \leq U), \ P \Rightarrow B, \ P \& (U=0) \{ C \} e: Q \]
\[ P \& (0 < U) \& (U \leq t) \{ C \} P \& (U < t) \]
\[ P \{ \text{while } B \text{ do } C \text{ od} \} e: Q \]  

(R9)

In order to prove that a loop terminates (normally or exceptionally) with a postcondition \( Q \),
the problem is no longer to establish by an inductive argument that the \( S_1, X_1 \) predicates of
\S5 convergent towards \( T \), but rather to find suitable invariant predicates \( P \) and variant
expressions \( U \). The induction is now hidden in rules (R8, R9).

The rules for empty, \( \triangleright e \), and block commands are:

\[ P \{ \} P \]  

(R10)

\[ P \{ \triangleright e \} e: P . \]

(R11)

\[ P \{ C \} x: Q \]

(R12)

\[ P \{ \text{begin } C \text{ end} \} x: Q \]

If \( Pr \) is a procedure introduced by the declaration

\[ \text{proc } Pr(v \ vp, w \ vrp, r \ rp) \ [EL] \ SDECL; C \]

then the proof rule for an invocation of \( Pr \) is

\[ P \{ \text{init} \} R, \ P \{ C \} x: Q, \ Q \{ \text{fin} \} R \]

\[ P \{ Pr(va,vra,ra) \} x: R \]

(R13)

where init, fin are as defined in \S5. Sequential composition is characterized by the rules:

\[ P \{ C1 \} e: Q \]

\[ P \{ C1;C2 \} e: Q \]  

(R14)

\[ P \{ C1 \} R, \ P \{ C2 \} x: Q \]

\[ P \{ C1;C2 \} x: Q \]  

(R15)
Finally, the proof rules for protected constructs are

\[
\frac{P \Rightarrow \neg B}{P \{ [B:C]\} P}
\]

(R16)

\[
\frac{P \Rightarrow B, \ P \{ C \{ x : Q \} \}}{P \{ [B:C]\} x : Q}
\]

(R17)

\[
\frac{P \{ S \} e : R, \ R \{ C_i \} x : Q}{P \{ S[EL_1:C_{i_1},...EL_n:C_{i_n}]\} x : Q}
\]

\((e \in EL_i)\)

(R18)

\[
\frac{P \{ S \} x : Q}{P \{ S[EL_1:C_{i_1},...EL_n:C_{i_n}]\} x : Q}
\]

\((x \notin \bigcup_{i=1}^{n} \text{EL}_i)\).

(R19)

The deductive system presented above has several important properties. Its proof rules are theorems of the wp semantics (their proofs are omitted for brevity reasons). So, the system is sound. Moreover, if \( P \Rightarrow \text{wp}(C,x,Q) \) is true, then \( P \{ C \} x : Q \) can be proved by using the rules (R1-R19). That is, the deductive system is complete. We do not give a detailed completeness proof; we only indicate how the most difficult case of this proof (the loop rules) can be handled.

**Theorem 4:** If \( P \Rightarrow \text{wp}(\text{while } B \text{ do } C \text{ od},;;Q) \) is true, there exists an assertion \( I \in \mathcal{L} \) and an integer expression \( U \) such that the following conditions are true:

\[
I \Rightarrow (0 \leq U), \quad I \& (U = 0) \Rightarrow \neg B, \quad I \& \neg B \Rightarrow Q,
\]

\[
I \& B \& (U \leq t) \Rightarrow \text{wp}(C,;;I \& (U < t)), \quad \text{and} \quad P \Rightarrow I.
\]

**Theorem 5:** If \( P \Rightarrow \text{wp}(\text{while } B \text{ do } C \text{ od,e,Q}) \) is true, there exists an assertion \( I \in \mathcal{L} \) and an integer expression \( U \) such that

\[
I \Rightarrow (0 \leq U), \quad I \Rightarrow B, \quad I \& (U = 0) \Rightarrow \text{wp}(C,e,Q),
\]

\[
I \& (0 < U) \& (U \leq t) \Rightarrow \text{wp}(C,;;I \& (U < t)), \quad \text{and} \quad P \Rightarrow I.
\]
Theorem 4 can be proved by explicitly constructing the required assertion I and expression U as follows:

\[ I \triangleq \bigvee_{i=0}^{\lfloor S \rfloor - 1} S_i, \text{ where } S_i, i \geq 0 \text{ are as defined in } \S 5 \]

\[ U \triangleq \text{if } S_0 \text{ then } 0 \text{ else} \]

\[ \text{if } S_1 \text{ then } 1 \text{ else} \]

\[ \quad \text{.} \]

\[ \text{if } S_{\lfloor S \rfloor - 1} \text{ then } |S| - 1 \text{ else } z \]

The verification that I and U satisfy the conditions required by Theorem 4 is a routine proof which we omit. The proof of Theorem 5 is similar. One constructs

\[ I \triangleq \bigvee_{i=1}^{\lfloor S \rfloor} X_i, \text{ where } X_i, i \geq 1 \text{ are as defined in } \S 5 \]

\[ U \triangleq \text{if } X_1 \text{ then } 0 \text{ else} \]

\[ \text{if } X_2 \text{ then } 1 \text{ else} \]

\[ \quad \text{.} \]

\[ \text{if } X_{\lfloor S \rfloor} \text{ then } |S| - 1 \text{ else } z \]

and shows that the conditions required by Theorem 5 are satisfied by these I and U.

Because of the very simple assertion and programming languages that we use, program correctness and robustness are decidable properties. Finally, one may remark that our deductive system is a conservative extension of a classical deductive system for proving the total correctness of one entry/one exit programs. In order to retrieve such a system, it is sufficient to delete any occurrences of the string "x:" in (R1, R2, R3, R7, R12, R13, R15) and merge the resulting rules with (R4, R8, R10).
7. AN EXAMPLE

In order to illustrate the use of the proof rules presented, let us verify that any invocation of the factorial procedure of Figure 3 satisfies the specifications:

\[(m<0) \{\text{FACT}(m,v)\}\neg: T\] \hspace{1cm} (F1)

\[(0\leq m) \& (m\not\in \text{MI}) \{\text{FACT}(m,v)\}\down{ow}: T\] \hspace{1cm} (F2)

\[(0\leq m) \& (m\in \text{MI}) \{\text{FACT}(m,v)\} v = m! .\] \hspace{1cm} (F3)

If we succeed in verifying that the body \(B(F)\) of the procedure satisfies the specifications

\[(n<0) \{B(F)\} \neg: T\] \hspace{1cm} (F4)

\[(0\leq n) \& (n\not\in \text{MI}) \{B(F)\}\down{ow}: T\] \hspace{1cm} (F5)

\[(0\leq n) \& (n\in \text{MI}) \{B(F)\} r = n! \] \hspace{1cm} (F6)

then by rules (R4, R13) it follows that (F1, F2, F3) are true. Moreover, since

\[(n<0) \lor (0\leq n) \& (n\not\in \text{MI}) \lor (0\leq n) \& (n\in \text{MI}) = T.\] \hspace{1cm} (F7)

we will also be able to conclude that the procedure FACT is robust. The body \(B(F)\) of FACT has the syntactic structure

\[B(F) \triangleq [n<0;\neg];C;L\]

where \(C \triangleq k:=0; r:=1\) always terminates normally. The loop \(L \triangleq \text{while } k<n \text{ do } LB \text{ od}\) has an ow exit point, defined by the occurrence of the "\text{ow}" sequencer in the loop body

\[LB \triangleq k:=k+1; r:=r*k\text{[ov;ow]}.\]

The truth of (F4) follows immediately from the rules (R17, R14, R1). Let us investigate the
case (F5) in more detail. By (R16) we have

\[(0 \leq n) \& (n \not\in \text{MI}) \{ [n < 0 : \text{neg}] \} (0 \leq n) \& (n \not\in \text{MI}) \quad (F8)\]

and from (F8) and rules (R4, R15) one can also conclude

\[(0 \leq n) \& (n \not\in \text{MI}) \{ [n < 0 : \text{neg}] \& C \} (0 \leq n) \& (n \not\in \text{MI}) \& (k = 0) \& (r = 1). \quad (F9)\]

If we can prove the truth of

\[(0 \leq n) \& (n \not\in \text{MI}) \& (k = 0) \& (r = 1) \{ \text{L} \}\text{ow: T} \quad (F10)\]

then by the sequencing rule (R15) we will be able to conclude from (F9, F10) that (F5) is true. The goal is now to prove (F10). Let us denote by \(a\) the greatest positive integer which has a machine representable factorial. We can rewrite the precondition of (F10) as:

\[P \triangleq (0 \leq a) \& (a < n) \& (a \not\in \text{MI}) \& ((a + 1) \not\in \text{MI}) \& (k = 0) \& (r = 1).\]

In designing the loop \(L\) the intention was to achieve the standard goal \(r = n!\) by incrementing the value of \(k\) from 0 to \(n\) while maintaining \(r = k!\) invariant. However, if \(P\) is true before the invocation of \(L\), then this first invariant approximation will not work, since for \(a < k \leq n\), the \(r = k!\) "invariant" will become false (because of the (f1) postulate). Thus, we have to tighten our invariant by bounding \(k\) by the largest positive integer with a machine representable factorial:

\[I \triangleq (0 \leq k) \& (k \leq a) \& (a < n) \& (a \not\in \text{MI}) \& ((a + 1) \not\in \text{MI}) \& r = k!\]

Since the distance between \(k\) and \(a\) decreases with each iteration, the form of \(I\) also suggests the following variant expression:

\[U \triangleq a - k.\]

The first two antecedents of (R9) are obviously true:
I \Rightarrow (0 \leq U), \ I \Rightarrow (k < n) . \quad (F11)

By rules (R4, R5, R15, R18, R19) we have

I \& (U=0) \{ k:=k+1; r:=r*k[ov:\textgreater ow]\} ow: T \quad (F12)
I \& (0 < U) \& (U \leq t) \{ k:=k+1; r:=r*k[ov:\textgreater ow]\} \ I \ & (U < t) . \quad (F13)

The truth of (F10) follows from (F11, F12, F13) by the loop rule (R9). Thus, the proof that the body B(F) of the procedure FACT satisfies the exceptional specifications (F4, F5) is terminated.

In order to prove that the actual standard semantics of B(F) is consistent with the specification (F6) one may first prove

(0 \leq n) \& (n \in \mathbb{N}) \{ SB(F) \} \ r = n! \quad (F6')

where SB(F) is obtained from B(F) by deleting all the text between square brackets. Then (F6) follows by the (R17, R19) rules instantiated with \( x = ; \). The proof of (F6') has been displayed so often in the literature that we will not repeat it.
8. CONCLUSIONS

Two kinds of desirable program properties have been investigated: correctness and robustness. We have argued that the notion of an exception is a useful structuring tool in developing correct and robust programs, in that it can simplify the tasks of specifying, implementing, verifying, and modifying such programs.

A programming language suitable for writing well-structured robust programs was presented. For simplicity, the only data type considered was (machine representable) integer. There seems to be no difficulty in including other machine representable data types like (finite) arrays, (finite) sets, etc., if the operations exported by these types are defined to be robust in the manner suggested in this paper. Issues underlying the specification, implementation and verification of programmer defined robust abstract data types are discussed in [1].

A sound and complete deductive system was proposed for proving (total) correctness and robustness properties of programs with exceptions. Since the exception mechanisms of [6,7] have a semantics close to ours, this proof system can be easily adapted to verify correctness and robustness properties of sequential ADA packages or CLU programs.

The verification conditions which are generated in program proofs were defined by induction on the syntactic structure of programs. This renders the verification condition generation process mechanizable and makes the proof system suitable for integration into a machine aided verification environment.

Although one would expect that the semantics of a programming language becomes more complex when one considers finite data types with exceptions instead of infinite data types, this does not seem to be so. First, infinite data types (like integers and stacks) still have inherent exceptions (like division by zero, stack underflow) so the problem of specifying
their exceptional meaning must be solved anyway. Second, by considering only finite data types, the underlying programming logic becomes simpler: one can use propositional logic instead of first order logic in order to reason about programs. There are more proof rules in a deductive system for verifying programs with exceptions than in a classical proof system for one entry/one exit programs. But the pay off is the reduction in the complexity of proofs which follows from the separation between the concerns for standard and exceptional program behavior.

Although the use of exceptions as a software structuring tool is especially valuable when dealing with larger programs (both at "programming in the small" and "programming in the large" levels), in this paper all the examples presented were very simple for pedagogical reasons. It is the purpose of a companion paper [3] to demonstrate the advantages of using the proposed language and deductive system in designing, understanding and reasoning about more realistic robust systems.

ACKNOWLEDGMENTS

The author would like to thank Eike Best, Jean-Pierre Black, Edsger Dijkstra, Joe Halpern, Tony Hoare, Brian Randell and John Tucker for helpful criticism and discussions on the subject of this paper.
REFERENCES


