Abstract

The normal purpose of a program execution is to produce from an initial state a final state, such that some intended relation holds between the two states. We discuss the exceptional situation in which the program is started in an initial state from which a final state violating the desired relation may be reached. This paper deals with the design of appropriate run-time checks, to be inserted in a program, for the detection of all possible exception occurrences. We show how such tests can be derived when forward and backward predicate transformer semantic definitions are available for the programming language. We also show an application to a realistic programming example.

Systematic Detection of Exception Occurrences

by

E. Best and F. Cristian

TECHNICAL REPORT SERIES

Series Editor: Mr. M.J. Elphick

Number 165
April, 1981

© 1981 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.
BEST, Eike,

Systematic detection of exception occurrences:
[By] Eike Best and Flaviu Cristian.

Newcastle upon Tyne: University of Newcastle upon Tyne,
Computing Laboratory, 1981.

(University of Newcastle upon Tyne, Computing Laboratory,
Technical Report Series. no. 165.)

**Added entries**

CRISTIAN, Flaviu
UNIVERSITY OF NEWCASTLE UPON TYNE.

**Suggested classmarks (primary classmark underlined)**

Library of congress:
Dewey (17th): 001.6425  001.6424
U, D.C. 681.322.06

**Suggested keywords**

ERROR DETECTION
PROGRAM DESIGN
PROGRAM SPECIFICATION
PROGRAM VERIFICATION

**Abstract**

The normal purpose of a program execution is to produce from an initial
state a final state, such that some intended relation holds between the two
states. We discuss the exceptional situation in which the program is started
in an initial state from which a final state violating the desired relation may
be reached. This paper deals with the design of appropriate run-time checks,
to be inserted in a program, for the detection of all possible exception
occurrences. We show how such tests can be derived when forward and backward
predicate transformer semantic definitions are available for the programming
language. We also show an application to a realistic programming example.

**About the author**

Mr. E. Best has been a Research Associate in the Computing Laboratory of the
University of Newcastle upon Tyne since June 1976.
Dr. F. Cristian has been a Research Associate in the Computing Laboratory of
the University of Newcastle upon Tyne since October 1979.
## Contents

1. Introduction

2. Intended and Actual Meanings of Programs
   2.1 Programming Language
   2.2 Specifications
   2.3 Relational Semantics
   2.4 Backward Specification-Transformer Semantics
   2.5 Forward Specification-Transformer Semantics

3. Run-Time Checks for the Detection of Exception Occurrences
   3.1 Preciseness of Run-Time Tests
   3.2 Unary Versus Binary Checks
   3.3 Checks in Conditionals and in Loops
   3.4 An Example: Bracket Matching

4. Conclusion

A. Appendices

References
1. Introduction

In proving the correctness of a program, a very common strategy is to consider only initial states in which certain properties are satisfied. For example, in the knowledge that a given array contains at least one positive element, one might prove a program for finding, say, the first positive element in that array, even though the program may otherwise (i.e. if the array does not contain any positive elements) lead to unpredictable results.

In practice there is however a strong demand for "robust" software, having a well-defined behaviour even in circumstances in which certain initial assumptions are no longer true. Such "exceptional" circumstances can occur whenever the inputs of the program cannot be guaranteed to have the properties they may be expected to have.

An example of this is a compiler where a syntactically well-formed program is expected as standard input, but where it cannot be guaranteed that all input programs are indeed well-formed. Another example would be a program requesting the exclusive use of a resource, expecting that at least one resource is free. In both cases provisions are needed for the treatment of unexpected (or exceptional) input. The need for a theory which can provide a basis for the systematic identification, detection and handling of exceptions has been expressed several times in the literature [GER78] [WUL80]. This paper explores ways of adapting previously developed semantic and correctness theories of programs [FLO67] [HOA69] [MAN74] [DIJ76] [DEB80] for the design of robust programs.

We focus specifically on two questions. Firstly, given a program and its specification, how can one characterise its standard and exceptional input domains? Secondly, how can one design appropriate run-time checks for the detection of any possible exception occurrence? We shall not deal with the question of actually handling an exception once it has been detected; for such a discussion the reader is referred to [CRJ81] and [LUC80].

Once the exceptional domain of a program is found, nothing would be
easier in principle than to test at the beginning of the program whether or not the initial state falls into the exceptional domain, thus making the program robust. However very frequently such a test would of necessity duplicate some of the work performed by the program itself, and it may therefore be much more natural and economical to insert tests within, or even at the end of, the program.

For instance, in order to test whether or not two given input values violate a bound restriction when added, one has to actually add them together. Similarly, it is the duty of a substantial part of a compiler to check whether or not an input program is well-formed, and it would be ridiculous to separate this checking entirely from the other tasks of the compiler. In this paper we derive verification conditions for robustness checks to be inserted anywhere in the program, as well as heuristic guidelines for actually choosing the place where to put such tests.

The paper is organised as follows. A mathematical framework integrating and generalising relational semantics, predicate transformer semantics and program correctness criteria is described in section 2, where our first question concerning exceptional input domains is also discussed. In section 3 we turn to our second question concerning the design of appropriate run-time checks for the detection of exception occurrences.

Sections 2.1 and 2.2 contain brief descriptions, respectively, of our programming language and our specification language. In sections 2.2-2.5 three equivalent types of semantics are defined, all of which are useful in deriving robustness tests. In section 3.1 we consider the sequential composition of two programs and the tests that can be inserted between them. In section 3.2 we discuss the circumstances in which such tests would have to involve the initial values of variables, and in section 3.2 we go on to consider checks in conditionals and iterative statements. Finally, in section 3.4 we apply all of this to the simplified but still practical example of a bracket matching program.
2. Intended and Actual Meanings of Programs

2.1 Programming Language

We use a modified version of guarded commands [Dijkstra76]. Our programs have the following general form:

\[ \text{<program>} ::= \text{<variable declarations>;}\text{<command>} \]

We assume that to every variable there is associated a specific set of values which it may take. We define the state of a program to be a mapping from the variables used in the program to their values. The set of all possible states is denoted by \( S \).

We use the following (hopefully self-explanatory) syntax for commands:

\[ \text{<command>} ::= \text{skip} | \text{abort} | \text{<assign>} | \text{<if>} | \text{<do>} | \]
\[ \text{<command>};\text{<command>} \]
\[ \text{<assign>} ::= \text{<var>} := \text{<expr>} \]
\[ \text{<if>} ::= \text{if } \text{<bool_1>} \text{ --> } \text{<command>} \text{ } \text{|} \text{| } \text{<command>} \text{ } \text{fi} \]
\[ \text{<do>} ::= \text{do } \text{<bool_1>} \text{ --> } \text{<command>} \text{ od.} \]

We consider only deterministic programs (i.e., in \text{<if>}, \text{<bool_1>}, ..., \text{<bool_n>} are mutually exclusive) but some of the formulas that follow remain valid in the non-deterministic case.

As we are interested in exceptional effects, we specifically allow expressions to be partial rather than total functions of variables. For instance,

\[ x := x + 1 \]

may be undefined if \( x = \text{MAXINT} \) initially. For any state \( s \), variable \( x \) and expression \( E \) we therefore introduce a predicate
defined \((x, E, s)\)

to denote the fact that in state \(s\), the evaluation of \(E\) will lead to a value that lies within the value domain of \(x\). This predicate will be used in the semantic definition of our language in Appendix A.2.

2.2 Specifications

We define the specification \(G\) (for "goal") of a program as a relation

\[
G \subseteq S \times S \tag{1}
\]

over the state space \(S\). \(G\) describes the intended effect of the program, the understanding being that for an initial state \(s^- \in S\), \((s^-, s) \in G\) if \(s\) could be a corresponding final state.

A specification may be non-deterministic in that several final states may correspond to a single initial state. We call \(G\) "undefined" for initial states to which no final state corresponds, and we define the domain of \(G\) as the set of initial states for which \(G\) is defined (for notation see Appendix A.1):

\[
\text{dom}(G) = \{s^- \in S \mid s^- G \neq \emptyset\} \tag{2}
\]

In practice a specification is not usually given by enumerating its member pairs, but rather by its characteristic binary predicate which for reasons of simplicity we also denote by \(G\):

\[
G : S \times S \rightarrow \{\text{true}, \text{false}\}, \quad \quad G(s^-, s) = \text{true} \iff (s^-, s) \in G.
\]

A specification can make reference to the value of a program variable, say \(x\), in the initial state \(s^-\) and in the final state \(s\). In order to simplify the notation, throughout the paper we adopt the convention of writing \(x^-\) instead of \(s^-(x)\) to denote the initial value of \(x\), and \(x\)
instead of s(x) to refer to the final value of x.

We use binary predicates involving both primed and unprimed variables in an analogous way in which unary predicates are used in [HOA69] and [DIJ76]. The difference is that the former represent relations over the state space while the latter represent subsets of the state space. A unary predicate can be seen as a special case of a binary predicate, involving either only primed variables ("precondition") or only unprimed variables ("postcondition").

As an example, assume that a program for the management of N resources (e.g. disk blocks) has to be written [CRI81], where N ≥ 1. The two services the program is to provide are: ALLOCATE some free resource and RELEASE a previously allocated resource. Suppose the variables are

\[
\text{var I: integer;}
\]
\[
\text{A: array (0..N-1) of 0..1,}
\]
and that for 0 ≤ j ≤ N-1, the j-th resource is free iff A[j] = 0.

If the intended effect of the ALLOCATE command is to assign to the variable I the name of a previously free resource then the (non-deterministic!) specification of ALLOCATE can be described by the following binary predicate:

\[
G(\text{ALLOCATE}) = (0 ≤ I ≤ N-1) \& (A'[I]=0) \& (A[I]=1) \quad (3)
\]

All initial states satisfying

\[
\exists j: 0 ≤ j ≤ N-1 \& A'[j]=0
\] (4)

have at least one corresponding final state making (3) true; and conversely, if (4) does not hold initially then (3) cannot be made true. In other words, (4) is (the characteristic unary predicate of) the domain of G(ALLOCATE). Input states violating (4) (i.e. states in which no resource is free) require exceptional treatment by the ALLOCATE
procedure.

2.3 Relational Semantics

Once the goal of a program has been stated as its specification, one has to compose it from the available primitives and hope—or prove—that it actually realises that goal. We define the actual meaning of a program $c$ again as a relation

$$R(c) \subseteq S \times S$$

over the state space. $R(c)$ can be defined by induction on the syntactic structure of $c$ using the semantic clauses given in Appendix A.2.

The interpretation of $R(c)$ is that $c$, started in an initial state $s^\prime$, must terminate in some $s \in s^\prime R(c)$. If it may fail to terminate when started in $s^\prime$, then $s^\prime R(c) = \emptyset$ [WAN77]. Perhaps this understanding appears unbecoming in that possible non-termination, infinite looping and aborting are all put into the same "bag" $s^\prime R(c) = \emptyset$. However, as we shall see it is perfectly possible to base our formalism on this understanding which is related to the wp formalism of [DIJ76], as shown below in section 2.4.

As already mentioned, we consider only deterministic programs. That is, we impose on $R(c)$ the condition

$$R^{-1}(c) \circ R(c) \subseteq Id$$

(see Appendix A.1). Thus, for deterministic $c$, $s^\prime R(c) s$ if and only if $c$, when started in $s^\prime$, terminates in $s$.

As an example, consider the program
which is supposed to implement the specification (3) above. In order to show this, one has to relate the concrete meaning \( R(ALLOCATE1) \) to the intended meaning \( G(ALLOCATE) \). One has to show that whenever the intended goal can be achieved a priori (i.e. (4) holds initially) then \( R(ALLOCATE1) \) actually yields a final state satisfying (3).

In general a command \( c \) will be said to implement a specification \( G \) iff

\[
\forall s' \in \text{dom}(G): \emptyset \subseteq s' R(c) \subseteq s' G. \tag{7}
\]

That is, for all inputs \( s' \) in the domain of \( G \) the program terminates \( (\emptyset \subseteq s' R(c)) \) and produces a final state satisfying \( G \) \( (s' R(c) \subseteq s' G) \). (7) corresponds to what is known as "total correctness" [MAN74].

The definition (7) is weak in the sense that it allows a deterministic program such as \( ALLOCATE1 \) to implement a non-deterministic specification such as \( G(ALLOCATE) \). It is also weak in the sense that nothing is required of the behaviour of \( c \) outside the domain of \( G \). We show in the next section how to prove that \( ALLOCATE1 \) implements the specification (3).

The next two sections introduce backward and forward semantics (the latter being a generalisation of the relational semantics just defined). We need both of these semantics in the determination of the run-time tests to be inserted in a program.
2.4 Backward Specification-Transformer Semantics

The well-known process of "backsubstituting a postcondition through the text of a program" [DIJ76] in order to derive properties of the program can readily be generalised for specifications as defined in section 2.2. The idea is to use relations over the state space not only as a means of globally describing a program c, but also as a means of describing the effect that components of c must have in order to ensure that c will indeed accomplish the overall goal.

For instance, if c is sequentially composed of c₁ and c₂ (see Figure 2), one is interested in the intermediate specification of c₁ needed to ensure that c₂ will accomplish the goal G:

![Diagram showing initial, intermediate, and final states with relations between them.]

**Figure 2**

In this situation we define the "weakest" specification for c₁ which guarantees that G is implemented by c = c₁;c₂ as a relation ws(c₂,G) ⊆ S x S satisfying

\[
s^- ws(c₂,G) t ⇔ \emptyset ⊆ tR(c₂) ⊆ s^- G. \tag{8}
\]

An initial state s⁻ and an intermediate state t thus stand in relation ws(c₂,G) iff c₂, when started in t, terminates in some final state s.
satisfying the global goal $s \in G$.

To see that $ws(c,G)$ as defined in (8) generalises the $wp$ (weakest precondition) semantics of [DIJ76], let $X \subseteq S$ be a subset of the state space and define

$$wp(c,X) = \{ s \in S \mid \emptyset \subseteq sR(c) \subseteq X\} \quad (9)$$

(compare also [WAN77]). Our interpretation of $R(c)$ implies that $wp(c,X)$ contains all initial states guaranteeing the termination of $c$ in a state in $X$, which is precisely the interpretation of the $wp$ operator in [DIJ76]. The connection between $ws$ and $wp$ can then be expressed as

$$wp(c, cod(G)) = cod(ws(c,G)).$$

What is different is that in our definition (8) both the second argument of $ws$ and $ws$ itself are considered binary rather than unary predicates.

The operator $ws$ has properties similar to those of $wp$. From (8) one derives

$$ws(c,\emptyset) = \emptyset \quad (10a)$$
$$ws(c,G_1 \cap G_2) = ws(c,G_1) \cap ws(c,G_2), \quad (10b)$$

and for deterministic programs satisfying (6) one also has

$$ws(c,G_1 \cup G_2) = ws(c,G_1) \cup ws(c,G_2). \quad (10c)$$

The $ws$ semantics of our programming language is given in Appendix A.2.

As an example of its use we prove the correctness of the program ALLOCATE1 (Figure 1) with respect to its specification $G(ALLOCATE)$ (formula (3) in section 2.2). This can be done by writing $G(ALLOCATE)$ at the end of the program and "mechanically" backsubstituting it through the program using the rules of Appendix A.2:
In each step of this backsubstitution one obtains from a given binary predicate a new binary predicate, again containing a mixture of primed and unprimed quantities; in fact, if $c$ denotes the statement in line $k$ in Figure 3 and $G$ denotes the specification in line $k+1$, then the specification in line $k-1$ equals $ws(c,G)$.

When the backsubstitution has come to an end (i.e. in line 0) one can identify initial and "current" states, simply by "priming" all unprimed variables. By this identification one obtains a unary predicate, namely the characteristic predicate of the set

$$st\_dom(c,G) = \{s^- \in S \mid s^- ws(c,G) s^-\} \quad (11a)$$

containing precisely those initial states for which $c$ is guaranteed to terminate in a final state satisfying $G$.

The set (11a) is by definition (8) a subset of $dom(G)$. All initial states $s^-$ outside $st\_dom(c,G)$, i.e. in

$$ex\_dom(c,G) = S \setminus st\_dom(c,G) \quad (11b)$$

must be treated as exceptional w.r.t. the given specification $G$ and program $c$. We therefore call $st\_dom(c,G)$ the "standard domain" or the
"implementation domain" of c with respect to G, and its complement, i.e. \( \text{ex\_dom}(c,G) \), is called the "exceptional domain".

The set \( \text{st\_dom}(c,G) \) equals the domain of G iff c implements G as defined in formula (7); formally,

\[
c \text{ implements } G \iff \text{dom}(G) = \text{st\_dom}(c,G) \tag{12}
\]

which can easily be proved from the definitions. In our example (Figure 3), if "priming" is actually applied to the specification in line 0, we obtain the characteristic predicate (4) of the domain of G; which, by (12), proves the correctness of ALLOCATE1.

2.5 Forward Specification-Transformer Semantics

Instead of asking for the weakest specification for \( c_1 \) which guarantees that \( c = c_1 ; c_2 \) implements some specification G, one could also ask for the strongest transition (abbreviated "st") relation which can be derived for c by knowing that the component command \( c_1 \) implements a specification \( G_1 \):

![Diagram](image)

**Figure 4**
\[ s' \triangleq \text{st}(c_2, G_1) s = \exists t: s' G_1 t \land t R(c_2) s \]

i.e., \( \text{st}(c_2, G_1) = G_1 \circ R(c_2) \) \hspace{1cm} (13)

The strongest "post"-specification of \( G_1 \) is thus simply the relational composition of \( G_1 \) and \( R(c_2) \).

Using (13), the correctness of \( c \) with respect to \( G \) can be established by proving that

\[ \forall s' \in \text{dom}(G): \emptyset \subseteq s' \text{st}(c, \text{Id}) \subseteq s' G \] \hspace{1cm} (14)

For example, by using the rules for \( \text{st} \) (see Appendix A.2), the fact that \texttt{ALLOCATE1} implements \( G(\text{ALLOCATE}) \) can be established by "pushing" the relation \( \text{Id} \) forward through the program as follows:

\[
0 \quad \{\text{Id}\} \\
1 \quad I := 0; \\
2 \quad \{I=0\} \\
3 \quad \textbf{do} \quad (I<N-1) \land (A[I]=1) \rightarrow I := I+1 \hspace{2pt} \textbf{od}; \\
4 \quad \{I=N \lor (0<I<N-1 \land A'[I]=0 \land A[I]=0)\} \land \\
\quad (\forall j: 0<j<I \rightarrow (A'[j]=1 \land A[j]=1)) \\
5 \quad A[I] := 1 \\
6 \quad (0<I<N-1) \land (A'[I]=0) \land (A[I]=1) \land \\
\quad (\forall j: 0<j<I \rightarrow (A'[j]=1 \land A[j]=1))
\]

Figure 5

As the last relation is non-empty and implies \( G(\text{ALLOCATE}) \), we have established again that the command \texttt{ALLOCATE1} correctly implements its specification. We shall in the next section use a similar method to derive exceptional tests.
3. Run-Time Checks for the Detection of Exception Occurrences

3.1 Preciseness of Run-Time Tests

Consider a specification G and a program c. We have seen that (whether or not c implements G) the set of input states S can be partitioned into the set of "standard inputs" st_dom(c,G) on the one hand and its complementary "exceptional domain" ex_dom(c,G) on the other hand (cf. formulae (11a) and (11b) in section 2.4). For inputs in the standard domain (which may be empty), c does implement G whereas for inputs in the exceptional domain (which may of course also be empty), c may either not terminate properly or end up in a final state not satisfying G. Moreover, if (and only if) the standard domain equals the entire domain of G then c does correctly implement G, as defined in formula (7) in section 2.3.

Exception handling aims at extending a program c in such a way that whenever c is started in an initial state outside the standard domain, then during its execution "a warning bell will ring", i.e. a specially designed piece of program (an "exception handler") becomes activated. We call such extended programs "robust" or "tolerant to exception occurrences" [CRI80]. The main purpose of exception handling is thus to make programs "total", or well-defined for all inputs.

In order to find out whether or not the input state is exceptional, there has to be a test somewhere during the execution of the program. This test does not necessarily have to take place at the beginning. For instance, in the program ALLOCATE1 (see Figure 1), rather than to test the exceptional condition

\[ \forall j: 0 \leq j < n-1 \Rightarrow A^+ [j] = 1 \]  

(15)

at the beginning of the program, it is more economical, as well as more natural, to place the test after the loop as follows:
1 \quad \text{I := 0;} \\
2 \quad \text{do (I < N-1) \& (A[I] = 1) \rightarrow I := I + 1 \text{ od};} \\
3 \quad [I > N-1 \rightarrow "\text{exception handler"}] \\
4 \quad A[I] := 1 \\

\textbf{Figure 6}

The meaning of the exceptional clause in line 3 is that the Boolean expression \( I > N-1 \) is evaluated and if found false, execution continues with the next statement; if found true, control is given to the exception handler whose exact working does not concern us here. We do not worry either about what precisely "economical" means in the context of tests; intuitively, one might say that predicates not involving quantifiers (such as \( I > N-1 \) in Figure 6) should normally be considered as more economical than predicates which do involve quantifiers (such as (15)).

We can easily convince ourselves that in line 2 in Figure 6, the test "\( I > N-1 \)" evaluates to "true" if and only if the initial state lies in the exceptional domain (15). For this reason we call it a "precise run-time test": it activates the exception handler when, and only when, an exception occurs.

In this section we characterise the precise tests that can be inserted in a program. It is important that such tests be precise because if they are too strong then it is possible that no exception handler is activated even though the input is exceptional, and if they are too weak then certain acceptable input states may find themselves treated as cases of failure.

Before defining precise tests formally, we make a few observations. The first is that the requirement for a test to be precise may not uniquely determine that test. For instance, the line

\[ 3' \quad [I = N \rightarrow "\text{exception handler"}], \]
if inserted instead of line 3 in Figure 6, would also "catch" precisely all exceptional inputs. This is due to the fact that the relation \( k < N+1 \) is part of the strongest postcondition after the loop (see Figure 5).

Our next observation is that not all locations in a program are equally appropriate in supporting exception detection. For instance, there is no way of catching an exception after the last assignment \( A[I] := 1 \) in Figure 6, because the exception will by this time already have led to an array bound violation when the expression \( A[I] \) is evaluated in line 4 with \( I = N \).

Our third (and last) observation is that if one allows tests to refer also to the initial state rather than just to the current state then in some cases exceptional tests could be inserted where otherwise they could not. As an example, we consider the following alternative implementation of \( G(ALLOCATE) \) which differs from \( ALLOCATE1 \) in the loop condition:

\[
ALLOCATE2 = I := 0;
\]
\[
do (I < N-1) \& (A[I] = 1) \rightarrow I := I + 1 \od;
\]
\[
[A[I] = 1 \rightarrow "exception handler"]
\]
\[
A[I] := 1
\]

Figure 7

Again, \( A[I] = 1 \) can be shown to be a precise exceptional test. By allowing the test to make reference to initial values (by using primes) one can in \( ALLOCATE2 \) (in contrast to the previous program) insert a precise test at the end of the program:
\[ I := 0; \]
\[ \text{do } (I < N - 1) \land (A[I] = 1) \rightarrow I := I + 1 \text{ od}; \]
\[ A[I] := 1; \]
\[ \text{[A}^\prime[I] = 1 \rightarrow "exception handler"\text{]} \]

\textbf{Figure 8}

Although in this example there is little point in doing so, for generality and uniformity we allow tests to refer to the initial states, as advocated, for example, in [HOR74]. Thus, we define a test $T$ formally as a binary predicate

$$ T : S \times S \rightarrow \{\text{true}, \text{false}\} \quad (16) $$

where the first $S$ refers to the initial state and the second $S$ to the current state. We return to the question of binary versus unary tests in the next section.

Consider now a general sequential decomposition of a program $c$:

$$ c = c_1; c_2. $$

We wish to determine, (a) whether a precise exceptional test can be inserted between $c_1$ and $c_2$, and (b) if so, to characterise the set of all such tests.

Question (a) is easy to answer. It has to be ensured that for every exceptional initial state, control actually reaches the point between $c_1$ and $c_2$ (unless there are different kinds of exceptions, a case which will be discussed below); in other words the following relation must hold:

$$ \text{ex\_dom}(c, C) \subseteq \text{wp}(c_1, S) \quad (17) $$
(where wp is as defined in (9)).

As to question (b), let T denote the test to be inserted between \( c_1 \) and \( c_2 \). We call T a "precise exceptional test" iff the following holds:

\[
\forall (s', t) \in R(c_1): s' \quad T \quad t \iff \text{not } s' \quad \text{ws}(c_2, G) \quad t \quad (18)
\]

Relation (18) expresses formally our intuitive understanding that the test T should evaluate to "true" if and only if the pair of states \((s', t)\) under consideration cannot be guaranteed to lead to a final state satisfying G.

With the two definitions

\[
T_s = R(c_1) \land \text{not } \text{ws}(c_2, G) \quad (19a)
\]

\[
T_w = \text{not } R(c_1) \lor \text{not } \text{ws}(c_2, G) \quad (19b)
\]

the formula (18) can be rewritten as the equation

\[
R(c_1) \land T = T_s \quad (20)
\]

and (20) can further be equivalently reformulated as a set of two implications

\[
T_s \implies T \quad (21a)
\]

\[
T \implies T_w \quad (21b)
\]

(21a) and (21b) imply that the set of precise tests equals the sublattice between \( T_s \) and \( T_w \) of the lattice of binary predicates, so that one is justified in calling \( T_s \) the "strongest precise test" and \( T_w \) the "weakest precise test". The following picture is a representation of the relationship between \( T_s \) and \( T_w \):
The formulae (21) can be interpreted as follows. Formula (21a) means that all pairs of (initial state, current state) such that $c_2$ is not guaranteed to establish $G$ must imply the truth of $T$; thus $T$ must be weak enough actually to activate an exception handler in case of an exception occurrence. $T_s$ itself may be too strong in the sense that too much is tested (an example of this will follow). Formula (21b), on the other hand, means that the truth of $T$ must imply either something impossible (not $R(c_1)$) or that $c_2$ cannot be guaranteed to establish $G$; thus $T$ must be strong enough not to treat any acceptable state as an exception. $T_w$ itself may be too weak in the sense that it may "catch" a lot of exceptional situations which never occur (again, an example follows).

Because $st(c, Id) = R(c)$ for all programs $c$ (see formula (12) in section 2.5), a general method of deriving precise tests $T$ is to backsubstitute (as exemplified in section 2.4) the specification $G$ through $c_2$ up to the point where $T$ is to be inserted and simultaneously to push (as exemplified in section 2.5) the relation $Id$ forward through $c_1$ up to the same point. Where the two meet we can form their difference (19a) to derive the strongest test $T_s$, or the union (19b) to derive the weakest test $T_w$, and we can use (20) as a verification.
condition for an arbitrary test \( T \) to be precise.

We illustrate this method on our example. Let us choose the following partitioning of \textsc{Allocate}:

\[
\begin{align*}
c_1 &= I := 0; \\
&\quad \text{do (I<N-1) \& (A[I]=1) \rightarrow } I := I+1 \text{ od;} \\
&\quad (*) \\
c_2 &= A[I] := 1 \\
\end{align*}
\]

\textbf{Figure 10}

and assume a test is to be inserted at point (*)

We first note that (17) is satisfied because \( c_1 \) always terminates. We then derive (compare Figure 3, line 4 and Figure 5, line 4):

\[
\begin{align*}
R(c_1) &= \text{st}(c_1, Td) = \\
&= (I=N \text{ or } (0 \leq I < N-1 \& A^-[I]=0 \& A[I]=0)) \& \\
&\quad \left( \forall j: 0 \leq j \leq I \rightarrow (A^-[j]=1 \& A[j]=1) \right) \\
&= (22a) \\
\end{align*}
\]

\[
\begin{align*}
\text{ws}(c_2, G(\text{Allocate})) &= \\
&= (0 \leq I < N-1) \& (A^-[I]=0) \\
&= (22b) \\
\end{align*}
\]

States satisfying (22a) but not (22b) are described by

\[
T_s = (I=N) \& \left( \forall j: 0 \leq j < N-1 \rightarrow (A^-[j]=1 \& A[j]=1) \right) \\
\quad (23a)
\]

and states violating either (22a) or (22b) by

\[
T_w = (I<0) \lor (I>N-1) \lor (A^-[I]=1) \lor (A[I]=1) \\
\quad \left( \exists j: 0 \leq j \leq I \& A^-[j]=0 \& A[j]=0 \right) \\
\quad (22b)
\]

Examining (23) we see that both in \( T_s \) and in \( T_w \) some terms are
redundant. The second term in $T_s$, for instance, is implied by the first term "I=N" in combination with (22a). Similarly, (22a) implies that the first term in $T_w$ can never be true. All in all, we have thus shown that $I=N$ and $I>N-1$ are precise tests, and that $I=N$ is the strongest non-redundant precise test that can be inserted at point (*) in Figure 10.

We also mention the following theorem whose proof is omitted for reasons of brevity. For deterministic programs,

$$T_s = \text{st}(c_1, \text{Id} \& \neg \text{ws}(c, C))$$

(24)

This means that instead of simultaneously backsubstituting C and pushing Id forward until they meet between $c_1$ and $c_2$, one can also backsubstitute C to the beginning of the whole program, negate the resulting predicate, identify "initial state" and "current state" (Id & ...), and push the result forward through $c_1$. This may sometimes be simpler than the other method.

We end this section with two remarks. Firstly we note that the condition (17) gives an indication about the location to choose for $T_s$. Since $\text{st\_dom}(c, C) \leq \text{wp}(c_1, S)$ by definition, (17) means nothing less than that $c_1$ is required to terminate for all inputs. Thus the tests must not be inserted "too late" in the program. Note that there always exists at least one decomposition of C in which $c_1$ terminates, namely the trivial $c = \text{skip;}c$. The reader is invited to derive the special cases of formulae (17)-(21) for $c = \text{skip;}c = c;\text{skip}$.

The second remark is that it is often natural for the exceptional domain $\text{ex\_dom}(c, C)$ to be partitioned into further subsets $E_1, E_2, \ldots$ etc. If these subsets are mutually disjoint then our formulae can easily be generalised. All that needs to be done is to refine the definition of "precise test" to that of a test T being "precise for exceptions in $E_1$" and to change (17)-(21) appropriately. This question will be discussed further in section 3.4.
2.2 Unary Versus Binary Checks

In this section we discuss under which circumstances a binary rather than a unary exceptional test is required. It is desirable that the test be unary rather than binary because otherwise the initial values of variables would in some way have to be kept saved in store.

Let us reconsider the program ALLOCATE2 of Figures 7 and 8:

\[
\begin{align*}
I & := 0; \\
\text{do } & (I < N-1) \& (A[I]=1) \rightarrow I := I + 1 \text{ od}; \\
A[I] & := 1
\end{align*}
\]

(*)

Figure 11

A precise test \(A[I]=1\) involving the initial value of \(A\) can be inserted at (*). However no precise test involving just the current state of \(A\) (which in this case is also the final state) can be inserted there.

The reason that no unary test can be inserted at (*) is the existence of two different initial states \(s', s''\) leading to the same final state \(t\) at (*), such that the pair \((s', t)\) satisfies the goal but \((s'', t)\) does not. To see this, define \(s'\) such that \(s'(A[N-1])=0\) and \(s'(A[j])=1\) for all \(0 \leq j < N-1\) (in which case \((s', t)\) happens to satisfy the goal \(G(ALLOCATE)\)), as compared to \(s''\) such that \(s''(A[j])=1\) for all \(0 \leq j < N-1\) (in which case the state at (*) is the same \(t\) as before, but \((s'', t)\) does not satisfy \(G(ALLOCATE)\)).

Generally, we define for \(c = c_1; c_2\) the following condition:

There are no two initial states \(s', s''\) and current state \(t\) s.t. (25)
\[
(s', t) \in R(c_1), \quad (s'', t) \in R(c_1), \quad tR(c_2) \subseteq s'G \quad \text{and} \quad tR(c_2) \nsubseteq s''G.
\]

It is possible to show that if, and only if, (25) holds then the test between \(c_1\) and \(c_2\) can be unary rather than binary. This gives a second indication of where to put the test: namely, the partitioning \(c_1; c_2\)
should be chosen such that (25) holds. We are not considering the question further whether the rather cumbersome property (25) can be made equivalent to, or at least a consequence of, a "nicer" property.

3.3 Checks in Conditionals and in Loops

In this section we derive analogues of the verification conditions (17)-(21) for precise tests in conditionals and in loops. For conditionals there is little to define. One has to ensure that the "if clause"

\[
\text{if } B_1 \rightarrow c_1 \quad \ldots \quad B_n \rightarrow c_n \quad \text{fi}
\]

cannot abort due to the non-existence of a true \( B_i \), and that every chosen alternative accomplishes the overall goal. Hence with

\[
T = \text{not } B_1 \quad \& \quad \ldots \quad \& \quad \text{not } B_n \quad \text{and}
\]

\[
T_i = \text{not } ws(c_i, G),
\]

the modified program

\[
[T \rightarrow \text{"exception handler"}] \quad \text{if } B_1 \rightarrow [T_i \rightarrow \text{"exception handler"}] \quad c_1 \quad \text{fi}
\]

\[
\ldots
\]

\[
[T_n \rightarrow \text{"exception handler"}] \quad c_n \quad \text{fi}
\]

is a robust version of the above "if clause".

On the other hand, we consider a loop
\[ c = \text{do } B \rightarrow c' \text{ od} \quad (26) \]

in which a test \( T \) is to be inserted such that the modified program

\[
\begin{align*}
\text{do } B \rightarrow [T \rightarrow \text{"exception handler"}] \\
& \quad c'
\end{align*}
\text{od}
\]

is a robust version of \( c \) with respect to a global goal \( G \). Again it has to be ensured (a) that control actually enters the body of the loop and (b) that \( T \) becomes true in some iteration iff the input was exceptional.

Property (a) can be ensured by postulating that \( B \) is true for every state in the exceptional domain, or in set notation that:

\[ \text{ex}_\text{dom}(c,G) \subseteq B \quad (27) \]

(27) is an analogue of (17) for loops.

Property (b) can be analysed as follows. We "cut" the loop (26) between \( B \) and \( c' \), i.e. at the point at which the test \( T \) is to be inserted. Let us define the states at this point to be the "intermediate" states. The transition relation giving the set of intermediate states reachable from a given intermediate state by an unspecified number of repetitions of the loop can be described in general by (for notation see Appendix A.2):

\[ R_0 = (R(c') \circ R(B))^* \quad (28a) \]

On the other hand, the relation between initial states and intermediate states and the relation between intermediate states and final states can be described, respectively, by the following two relations \( R_1 \) and \( R_2 \):

\[ R_1 = R(B) \circ R_0 \quad (28b) \]
\[ R_2 = R_0 \circ R(c') \circ R(\text{not } B) \quad (28c) \]
Pictorially, we may represent the relationship between these relations as follows:

![Diagram showing relationships between relations R₁, R₀, and R₂ with intermediate state t]

**Figure 12**

Property (b) can be ensured iff for all intermediate states t for which R₂ is not guaranteed to satisfy the overall goal G, another intermediate state, say x, is reachable from t such that the test T will hold in x. This underlies the following definition. We call a (binary) test T "precise" for (26) iff

\[
\forall (s', t) \in R₁: \not s' \not\dashv ws(R₂, G) \ t \Rightarrow \exists x \ s' \dashv R₀ \ x \ & s' \dashv T \ x
\]

(29)

where \( ws(R, G) \) for two relations R and G is the obvious extension of formula (8) in section 2.4:

\[
s' \dashv ws(R, G) \ t \iff \ \emptyset \subseteq tR \subseteq s'G.
\]

Formula (29) is the analogue of (18) for loops. Using relational algebra, (29) can be rewritten as the equation

\[
R₁ \ & \not\dashv ws(R₂, G) = R₁ \ & (T \circ R₀^{-1})
\]

(30)
Formula (30) is the analogue of (20) for loops. Since, by our overall assumption (6), $R_0$ is deterministic, $R_0^{-1}$ is injective (see Appendix A.1). Using this fact it may be proved that if two tests $T_1$ and $T_2$ satisfy (30) then so do $T_1 \& T_2$ and $T_1 \lor T_2$. Hence the set of tests satisfying (30) is again a sublattice of the lattice of binary predicates.

As an example we prove that the test $I=\text{N-1}$ in the following alternative implementation of $G(\text{ALLOCATE})$ is again a precise test:

\[
\begin{align*}
\text{ALLOCATE3} & \equiv I := 0; \\
& \quad \text{do } A[I] := 1 \rightarrow [I=\text{N-1} \rightarrow "\text{exception handler}"] \\
& \quad \quad \quad I := I+1 \\
& \quad \text{od}; \\
& \quad A[I] := 1
\end{align*}
\]

Figure 13

(27) is fulfilled because, taking into account the initial assignment of 0 to $I$, the truth of the exceptional condition

\[
\forall j: 0 \leq j < \text{N-1} \Rightarrow A^e[j] = 1 \quad (15)
\]

implies in particular that $A[I] = 1$ on loop entry.

To prove (30), we consider an initial state $s^e$ and an intermediate state $t$. We denote by $I^t = t(I)$ the value of $I$ in state $t$. $(s^e, t) \in R_1$ implies that

\[
I^t \in \{0, \ldots, \text{N-1}\} \land \forall j: 0 \leq j \leq I^t \Rightarrow A[j] = 1; \quad (31a)
\]

On the other hand, not $s^e$ ws $(R_2, G)$ $t$ means that

\[
\forall j: I^t < j < \text{N-1} \Rightarrow A[j] = 1 \quad (31b)
\]

25
For another intermediate state \( x \), \( s' \) \( T \) \( x \) means that the value of \( I \) in state \( x \) equals \( N-1 \), and \( T R_0 x \) then means that
\[
\forall j: 1 \leq j \leq N-1 \Rightarrow j \in \{0, \ldots, N-1\} \land A[j] = 1
\]
(31c)

This proves (30) because under the assumption that the first clause in (31a) holds, (31b) and (31c) are equivalent.

We end this section with two remarks. It might be objected that the procedure outlined in the last paragraph breaks down if, say, the search of the array \( A \) is begun with the index \( 1 \) rather than \( 0 \), because in this case the initial state in which the very first, but no other, resource is free (\( A[0] = 0 \) and \( A[j] = 1 \) for \( 1 \leq j \leq N-1 \)) will give rise to an exception. However a program in which the search begins with \( 1 \) does not itself implement the specification \( \mathcal{G}(\text{ALLOCATE}) \), as defined in sections 2.2 and 2.3. The initial state just mentioned would therefore belong to the exceptional domain rather than to the standard domain, which justifies the detection of an exception.

Our second remark concerns the design of the last version of the allocation algorithm. In contrast to the previous two versions, in \textsc{ALLOCATE3}, the tests on \( A \) and on \( I \) are separated out in such a way that all testing on \( I \) occurs just immediately before \( I \) is actually changed. This seems indeed the most natural location for the test to be placed, because the inside of the loop can now be regarded as an "action on \( I \)" by itself, indicating proper program structure. This leads not only to a (however slight) gain in the average amount of testing done, but also to the desirable property that \( I \) never actually assumes any values outside its "natural" domain between \( 0 \) and \( N-1 \).

It may well be possible that this line of reasoning can be generalised to derive a third, albeit more heuristic, indication of where to put tests in a program; namely, to try and separate them out in such a way that the test of a variable and the actual change of this variable are as near together as possible. We shall return to this point in the next section.
3.4 An Example: Bracket Matching

The purpose of this section is to illustrate an application of our formalism to a non-trivial example (which may be familiar to compiler writers). Assume a stream $W$ of characters is given as an input to a Bracket Matching (BM) program. Some of the characters are 'B' (for "begin") and some of them are 'E' (for "end"). The string $W$ may, for example, represent the internal encoding of an Algol program; the program BM is then required to analyse the block structure of that program by finding all matching "begin"-"end" pairs.

In this situation the following is assumed (see Figure 14). The first character in $W$ is a 'B', and in the part of $W$ between the first 'B' and its matching 'E' a "space" is provided after every 'B' for the index of its matching 'E' to be inserted; furthermore, in that part of $W$, all other characters must remain unchanged, while the characters following the last 'E' are irrelevant and may be destroyed. We further assume that the depth of nesting of 'B'-'E' pairs does not exceed a positive constant $D > 0$ (reflecting, for example, dynamic memory allocation constraints imposed by a host operating system).
Let the input stream $W$ be represented by an array $W$ containing integer encodings for characters 'E', 'B' etc (we assume 0 is the encoding for the "space"):

\[ \text{var } W: \text{array}(1..N) \text{ of integer}, \]

where $N>0$ is a positive constant. Let $nb(K,i,j)$ be the number of occurrences of an arbitrary integer constant $K$ between two indices $i$ and $j$ (inclusively) of $W$, $1 \leq i \leq j \leq N$:

\[ nb(K,i,j) = \text{card}\{1 \mid i \leq j \& W[1]=K\} \]

Then

\[ D(i,j) = nb(\text{'B'},i,j) - nb(\text{'E'},i,j) \]

denotes the difference between the number of 'B's and the number of 'E's between $i$ and $j$. Furthermore let for $1 \leq i \leq N$
Candidates\( (i) = \{ j \mid 1 \leq j \leq N \land W^*(j) = ^*E^* \land D(i,j) = 0 \} \)

denote the set of indices of \(^*E^*\)'s between \(i+1\) and \(N\) which are candidates for matching a \(^*B^*\) at position \(i\), and let for \(1 \leq i \leq N\)

\[
\text{Match}(i) = \begin{cases} 
\text{if } W^*[i] = ^*B^* \land \text{Candidates}(i) \neq \emptyset \\
\text{then } \min(\text{Candidates}(i)) \\
\text{else } N+1
\end{cases}
\]

be the index of the matching \(^*E^*\) of a \(^*B^*\) located at position \(i\) in \(W\) (the value \(N+1\), which is not an index in \(W\), means "no-matching").

The formal specification of the EM program as a relation between initial and final states can be stated as a conjunction of four simpler constraints:

\[
G(EM) = P1 \land P2 \land P3 \land P4
\]

where

\[
P1 = (W^*[1] = ^*B^*) \land (\text{Candidates}(1) \neq \emptyset)
\]

requires that the first character must be a \(^*B^*\) and that a matching \(^*E^*\) must exist in \(W\).

\[
P2 = \forall i: (1 \leq i < \text{Match}(1)) \land (W^*[i] = ^*B^*) \implies \\
(W^*[i+1] = 0) \land (W[i+1] = \text{Match}(i))
\]

requires that matching indices have to be inserted after each \(^*B^*\) in place of the "space" reserved for this purpose.

\[
P3 = \forall i: (1 \leq i < \text{Match}(1)) \implies \ (D(i,1) < D)
\]

limits the depth of possible nesting of \(^*B^*-^*E^*\) pairs.

\[
P4 = \forall i: (1 \leq i < \text{Match}(1)) \land (W^*[i] = W[i]) \implies \ (W^*[i-1] = ^*B^*)
\]

represents the requirement that between 1 and Match(1) all entries other than "spaces" must remain untouched.

One can remark that P1 imposes a direct constraint on \(W^*\): the first character in \(W^*\) must be a \(^*B^*\) and the set Candidates(1) must not be empty. This remark, together with a part of P2 (the constraint that each \(^*B^*\) has to be followed in \(W^*\) by a 0) and the whole of P2 leads us
to advance

\[ SD = (W[i] = 'B' \land \text{Candidates}(1) \neq \emptyset) \land \\
(\forall i: (1 \leq i < \text{Match}(1)) \land (W[i] = 'B') \Rightarrow W[i+1] = 0) \land \\
(\forall i: (1 \leq i < \text{Match}(1)) \Rightarrow D(1, i) \leq D) \]  

(32)

as a necessary condition for the existence of a final state \( W \) satisfying \( G(BM) \). The condition is also sufficient. Indeed, if Candidates(1) \neq \emptyset then one can prove that Match(i) \leq N+1 for every entry \( i \) such that \( 1 \leq i < \text{Match}(1) \) \& \( W[i] = 'B' \), and therefore it is possible to define a \( W \) with the "spaces" following the 'B's replaced by corresponding Match(i)'s (P2) and with all the other entries unmodified (P4).

Therefore (32) is the characteristic predicate of \( \text{dom}(G(BM)) \).

The implementation BM given in Figure 15 uses a running index I in \( W \) and a stack ST with pointer P for storing the indices of the "spaces" which follow 'B's and which have to be filled in by adequate matching indices of 'E's.
\texttt{var} \: I: \: 1..N; \\
P: \: 0..D; \\
ST: \: \texttt{array} \: 1..D \: \texttt{of} \: 1..N; \\
0 \: \: I := 2; \\
1 \: \: P := 1; \\
2 \: \: ST[P] := I; \\
3 \: \{0 \leq P = D(1, I) \text{ is a loop invariant}\} \\
4 \: \: \text{do} \: P \leq 0 \rightarrow I := I + 1; \\
5 \: \: \quad \text{if} \: W[I] = 'B' \rightarrow I := I + 1; \\
6 \: \: \quad \quad \: P := P + 1; \\
7 \: \: \quad \quad \: ST[P] := I \\
8 \: \: \quad \[ \] \: W[I] = 'E' \rightarrow \{1 \leq ST[P] \leq N \& W[ST[P]] = 0 \& \\
9 \: \: \quad \quad \quad \quad \quad \quad \quad \: W[ST[P] - 1] = 'B'] \\
10 \: \: \quad \quad \quad \quad \quad \quad \quad \: W[ST[P]] := I; \\
10 \: \: \quad \quad \quad \quad \quad \quad \quad \: P := P - 1 \\
\text{od.} \\

\textbf{Figure 15: BM Standard Algorithm}

In order to prove that the EM program implements $G(\text{BM})$, one has to establish that

\[ SD \subseteq \text{st-dom}(EM, G(EM)) \]  \hfill (33)

Because by (10b),

\[ \text{ws}(EM, P_1 \& P_2 \& P_3 \& P_4) = \text{ws}(EM, P_1) \& \ldots \& \text{ws}(EM, P_4) \]

one can consider the four terms of the specification separately. To aid the argument, two intermediate assertions have been inserted in lines 3 and 8 of Figure 15. (It has to be added that the correctness proof involves only the implication (33), and that parts of the assertions inserted in Figure 15 are true only under the assumption that $SD$ holds initially). The loop invariant is true initially because $SD$ implies
$W'[1] = \text{N} > 1$, and it is not difficult to show that it remains true over the loop. The assertion in line 8 is true before "popping" the stack because every "push" on the stack (lines 5-7) ensures its truth. The convergence of the loop is a consequence of the fact that under the assumption SD, Match(I) N and the running index I successively "visits" all array elements between 2 and Match(I), and that if I = Match(I) the loop invariant implies P = D(1, I) = D(1, Match(I)) = 0. Further details about the proof of (33) are omitted here.

Once the correct behaviour of the program within the domain of its specification has been established (32), one can concentrate on its behaviour outside this domain. We obtain the exceptional domain of $BM$ by negating (32):

$$\text{ex}_\text{dom}(BM, G(BM)) = E_1 \vee E_2 \vee E_3 \vee E_4$$

(34)

where

$$E_1 = (W'[1] \neq N)$$

corresponds to the case when the first character is not a $N$.

$$E_2 = (\text{Candidates}(1) = \emptyset)$$

corresponds to the case when no matching $N$ exists for the first $N$ in W.

$$E_3 = (\exists i: (1 \leq i < \text{Match}(1)) \& (W'[i] = N) \& (W'[i+1] \neq 0)$$

corresponds to the case where a $N$ is not immediately followed by a "space" as required.

$$E_4 = (\exists i: (1 \leq i < \text{Match}(1)) \& (D(1, i) > D)$$

corresponds to the level of nesting exceeding the limit D.

When designing a set of tests for $BM$ to detect all possible exceptions in (34), we apply the principle mentioned in section 3.3, namely to separate the tests out in such a way that they correspond in a clear way to the variables being tested. More concretely, we endeavour to make sure that none of the local variables in $BM$ will ever assume a value outside the domain specified in its declaration; for example, whenever I is to be incremented, a test whether I = N will be inserted. Our full solution is shown in Figure 16 below.
The test on \( W \) in line 11 does not satisfy the above criterion since it is separated from the place in which \( W \) is actually changed, namely line 16. However, its inclusion can be justified by the following argument. Instead of testing \( W \) in line 11, a test

\[
[W\{ST[P]\} \& 0 \rightarrow HE_3]
\]

could also have been inserted just before line 16. The test in line 11 can be seen as an optimised version of this latter test, implying the truth of the second term in the assertion in line 15 and thus making the second test superfluous.

0 \([W[1]=B \rightarrow HE_1]\)
1 \([N=1 \rightarrow HE_2]\)
2 \([W(2)+0 \rightarrow HE_3]\)
3 \(I := 2;\)
4 \(P := 1;\)
5 \(ST[P] := I;\)
6 \([0 \leq P = D(1,1) \text{ is a loop invariant}]\)
7 \(\text{do } P+0 \rightarrow [I=N \rightarrow HE_2]\)
8 \(I := I+1;\)
9 \(\text{if } W[I]=B \rightarrow [I=N \rightarrow HE_2]\)
10 \(\text{I := I+1;}\)
11 \([W[I]+0 \rightarrow HE_3]\)
12 \([P=D \rightarrow HE_4]\)
13 \(P := P+1;\)
14 \(ST[P] := I;\)
15 \(\text{if } W[I]=E \rightarrow \{I<ST[P]<N \& W[ST[P]]=0 \& W[ST[P]-1]=B\}\)
16 \(W[ST[P]] := I;\)
17 \(P := P-1\)

\text{od}

\text{Figure 16: Robust BM Program}
Intuitively, the exception handlers \( HE_1-HE_4 \) shown in Figure 16 correspond to the four exceptional sub-domains \( E_1-E_4 \) defined in (34). In the remainder of this section we clarify in what way they correspond to each other, and we also show that the total set of tests in lines 0, 1, 2, 7, 9, 11 and 12 of Figure 16 is precise for the exceptional domain (34), satisfying:

(i) whenever a test is true then an exceptional input state has been detected;
(ii) conversely, every exceptional input state leads to an intermediate state in which some test is true.

In proving this we encounter the following complication. It would be nice to show, using say (29), that for each individual exceptional domain \( E_i \), the corresponding exception handler \( HE_i \) becomes activated if and only if the input state was in \( E_i \). However, as remarked at the end of section 3.1, such a strategy is possible only for disjoint sub-domains.

In our situation we have four non-disjoint exceptional sub-domains in (34). As a result, if an input state is in two or more of those domains, it cannot a priori be determined which exception will be detected first (if at all). For example, if the input of the BM program shown in Figure 16 satisfies both \( E_3 \) and \( E_4 \), either \( E_3 \) or \( E_4 \) can be detected first, depending on the particular shape of \( W' \).

However, for every individual exceptional sub-domain we can prove a slightly weaker property than full preciseness, namely:

(i) if an exception handler \( HE_i \) is activated then the input state was in \( E_i \);
(ii') for every input state in \( E_i \), either the corresponding handler \( HE_i \) eventually becomes activated, or another exception \( E_j \) is detected prior to that (indicating that the input state lies in the intersection \( E_i \& E_j \)).

The fact that properties (i) and (ii') hold for \( E_1-E_4 \) in our example, justifies our earlier remark that the four exception handlers
$HE_1 - HE_4$ correspond to $E_1 - E_4$, respectively. Note that the distinction between $E_1 - E_4$ is also reflected in the state variables of EM: $E_2$ corresponds to a check on $I$, $E_4$ corresponds to a check on $P$, and $E_1$ and $E_3$ correspond to checks on $W$.

If (i) and (ii') can be proved for all (overlapping) exceptional sub-domains then the truth of (i) and (ii) follows directly for the union of these sub-domains; i.e. in combination the tests are precise for the full exceptional domain.

In the sequel we prove (i) and (ii') for the exception $E_4$ of our example. For $E_1 - E_3$, the demonstration is analogous. The proof which follows is not formal but, we hope, precise enough to avoid any misunderstanding.

(i) Assume that during the execution of the loop, an intermediate state has been reached such that the test $P=D$ of line 12 evaluates to true. From the loop invariant $P=D(l, I)$, which was true at the end of the previous iteration, and from the fact that the two most recently visited array elements contain 'B' and 0, respectively, (lines 9-11), we can infer that $D(l, I)=D+1$ holds for the actual value of $I$. As I runs between 2 and Match(1), it follows that $E_4$ was true initially.

(ii') Conversely, assume $E_4$ is true initially. Define

$$i_0 = \min\{i \mid (1 \leq i < \text{Match}(1) \land D(l, i) > D)\}$$

By this definition, $i_0$ always "points" to a 'B', i.e. $W[i_0]='B'" (otherwise the minimality of $i_0$ would be contradicted). As I is only incremented by 1 in EM, we have to consider the following two cases.

(a) During the execution of the loop, I reaches the value $i_0$. This can only be a consequence of executing the statements in line 8 (sub-case a1) or in line 10 (sub-case a2).

(a1) If $i_0 = N$ the guard of $HE_2$ in line 9 leads to the detection of $E_2$. If $i_0 < N$ and $W[i_0+1]='0'" the guard of $HE_2$ leads to

35
the detection of $E_3$. If $i_0 < N$ and $W[i_0+1]=0$ the exception $E_4$ is detected in line 12. No other possibilities remain.

(a2) Since $W[i_0] = \wedge$ the guard of $HE_3$ in line 11 is true and the exception $E_2$ is detected.

(b) I never reaches the value $i_0$ during the normal execution of the EM program. Two sub-cases have to be considered.

(b1) The loop is not entered. As the execution of the statements in lines 3,4,5 cannot lead to the occurrence of a language defined exception, it follows that one of the guards of the lines 0,1,2 was true and an exception $E_1$, $E_2$ or $E_3$ has been detected.

(b2) The loop is entered. Two outcomes are in principle possible.

(b21) The loop terminates normally. It follows that $Match(1) < N+1$. Since $I$ takes successively all the integer values between 2 and $Match(1)$ it follows that either $i_0 < 2$ or $i_0 > Match(1)$, which together with $D > 0$ contradicts the definition of $i_0$. Therefore (b21) cannot occur.

(b22) The loop terminates abnormally. As a result of their design, the tests in the exceptional clauses in Figure 16 provide a set of "local" arguments that none of the expressions and statements in the standard algorithm can lead to the occurrence of language defined exceptions. For instance, incrementing $P$ in line 13 cannot lead outside the domain $0..D$ of $P$ because of the immediately preceding test. Similarly, the well-definedness of the statement in line 16 can be inferred from the immediately preceding assertion; etc. It follows that the abnormal termination of the loop must be due to a guard for $HE_2$, $HE_3$ or $HE_4$ being true in some intermediate state.

Thus (i) and (ii') are established for $E_4$. Proving (i) and (ii') for all of $E_1-E_4$ establishes the preciseness of the total set of tests for the exceptional domain (34).
We point out that for the argument in case (b) above to be "easy", it has been important that every individual component statement in the program could easily ("locally") be examined to discover whether or not it would lead to the occurrence of a language defined exception. This in turn has been a consequence of our design decisions to place the test as near as possible to the point of change of a variable being tested. This also relates to the fact that the variables used in BM reflect in a natural way the form of the specification, i.e. it being a conjunction of sub-specifications, leading to a conjunctive form for the expression for the standard domain and to a disjunctive form for the exceptional domain (34).
4. Conclusion

Although the identification and detection of possible exception occurrences are important problems in the design of robust software [GER78], we do not know of any other attempt of establishing a framework both rigorous and practical for solving them. In practice, software designers rely upon their intuition and experience in dealing with them, and therefore the identification and detection of possible exceptional conditions is often just as (un)reliable as human intuition is.

The paper proposes a systematic approach for solving the above mentioned problems. It is a part of a larger effort aiming at providing a rigorous framework for the design of correct and robust software. The run-time checks to be inserted in programs were expressed in terms of the forward and backward specification transformer semantic definitions of a simple programming language. Important problems may arise in practice when determining such checks if the programs contain loops. Indeed, it is well known that the determination of such transformers for loops is usually a hard problem, requiring the use of inductive reasoning.

We have used binary predicates (relations) rather than unary predicates (subsets) for describing the intended meaning of programs. Another possible approach would be to duplicate the state space with "auxiliary" or "logical" variables which store the initial values of the "real" variables, and allow predicates to involve both "real" and "logical" variables, while the commands would be allowed to modify only the "real" variables. The two approaches seem to be in principle equivalent, since a specification (relation) in our approach would correspond to a unary predicate (subset) over the duplicated state space (i.e. the cartesian product) of the other approach.

The expression of the exact run-time checks were given under the simplifying assumption that the exceptional input domain is not partitioned. In practice however (as the Bracket Matching program or examples in [CRIP1] show) the exceptional domain is frequently partitioned into several exceptional sub-domains. The reason for such a partitioning is that the exception handlers which have to be activated
for initial states in different sub-domains have to be in general different.

Exact run-time checks for detecting exception occurrences when the exceptional domain is partitioned into disjoint sub-domains can be derived by using formulae like those given in this paper. However, the case when the exceptional domains overlap requires a modification of the proof method, whereby the formulae given have to be weakened for the sub-domains themselves. We have illustrated such a modification in the Bracket Matching example.

This paper should not be interpreted as a case for inserting run-time checks in a program whenever possible. Static semantic analysis [COU78] or proving methods should be used to discover or to prove that under certain assumptions some of the exception preconditions which have been identified for a program never become true at run-time. In such cases, of course, no run-time checks are needed and the overhead of checking for things that cannot happen should be avoided.

The approach presented here is of use in the other case, in which the assumptions about the environment in which a program will run cannot be used to guarantee the constant falsehood of an exception precondition. Only in those cases run-time checks for detecting the corresponding exceptions should be inserted. We therefore consider our approach to be complementary to those oriented towards proving the absence of exception occurrences.
A. Appendices

A.1 Relation Algebra

Throughout the paper, the connectives \& and \lor are not commutative; we define

\[ a \& b = \text{if } a \text{ then } b \]
\[ a \lor b = \text{if } a \text{ then true else } b. \]

However the \& and \lor connectives inherit associativity and most other properties of the classical logical connectives.

Let \( S \) be a set, \( x, y \in S \) and \( G, H \) binary relations over \( S \). The basic operations over binary relations are:

\[
\begin{align*}
\text{(binary)} & \quad x \circ G \circ H y = \exists z: x \circ G z \& z \circ H y \\
& \quad x \circ G \cup H y = x \circ G y \lor x \circ H y \\
& \quad x \circ G \cap H y = x \circ G y \& x \circ H y \\
& \quad x \circ G \setminus H y = x \circ G y \& \text{not} x \circ H y \\
\text{(composition)} & \quad (union) \\
\text{(intersection)} & \quad (difference) \\
\end{align*}
\]

\[
\begin{align*}
\text{(unary)} & \quad x \circ G^{-1} y = y \circ G x \\
& \quad x \circ G^* y = \exists n \geq 0: x \circ G^n y \\
& \quad \text{where } G^n \text{ is defined inductively:} \\
& \quad G^0 = \text{Id}, G^n = G \circ G^{n-1} \\
\text{(inverse)} & \quad (complement) \\
\text{(star)} & \quad (identity) \\
\end{align*}
\]

\[
\text{(nullary) Id: } x \circ \text{Id} y = x = y \\
\emptyset \quad \text{(empty relation)}
\]

Operations from relations into the set \( S \):

\[
\begin{align*}
\text{Dom}(G) &= \{ x \in S \mid \exists y: x \circ G y \} \quad \text{(domain)} \\
\text{Cod}(G) &= \{ y \in S \mid \exists x: x \circ G y \} \quad \text{(codomain)} \\
xG &= \{ y \in S \mid x \circ G y \} \\
Gy &= \{ x \in S \mid x \circ G y \}
\end{align*}
\]
Special classes of relations:

- **Deterministic** if \( G^{-1} \circ G \subseteq \text{Id} \)
- **Injective** if \( G \circ G^{-1} \subseteq \text{Id} \)
- **Surjective** if \( \text{Cod}(G) = S \)
- **Total** if \( \text{Dom}(G) = S \)

### A.2 Semantics of the Programming Language

#### Relational Semantics (see section 2.3):

\[
\begin{align*}
R(\text{skip}) &= \text{Id} \\
R(\text{abort}) &= \emptyset \\
R(x := E) &= \{(s',s) \mid \text{defined}(x,E,s') \land s(x) = \text{value of } E \text{ in } s' \land \forall y \neq x : s(y) = s'(y)\}
\end{align*}
\]

\[
R(\text{if } B_1 \rightarrow c_1 \ldots [ ] B_n \rightarrow c_n \text{ fi}) = R(B_1) \circ R(c_1) \ldots \circ R(B_n) \circ R(c_n)
\]

where \( R(B) = \{(s,s) \mid \text{defined}(B,s) \land B(s) = \text{true}\} \)

\[
R(\text{do } B \rightarrow c \text{ od}) = (R(B) \circ R(c))^* \circ R(\text{not } B)
\]

\[
R(c_1; c_2) = R(c_1) \circ R(c_2)
\]

(the last three formulae hold only for deterministic commands)

#### Weakest Specification-Transformer Semantics (section 2.4):

\[
\begin{align*}
\text{ws}(\text{skip},G) &= G \\
\text{ws}(\text{abort},G) &= \emptyset \\
\text{ws}(x := E,G) &= \text{Def}(x,E) \land G[E/x]
\end{align*}
\]

where \( \text{Def}(x,E) = \{(s,s) \mid \text{defined}(x,E,s)\} \)

and \( G[E/x] \) stands for the specification obtained from \( G \) by substituting all free occurrences of \( x \) by \( E \).

\[
\begin{align*}
\text{ws}(\text{if } B_1 \rightarrow c_1 \ldots [ ] B_n \rightarrow c_n \text{ fi},G) &= \exists j \in \{1, \ldots, n\} : B_j \land \forall j \in \{1, \ldots, n\} : B_j \Rightarrow \text{ws}(c_j,G) \\
\text{ws}(\text{do } B \rightarrow c \text{ od},G) &= \exists G_1
\end{align*}
\]
where \( G_0 = R(B) \land \neg G \),
\[ G_{i+1} = R(B) \land \text{ws}(c, G_i). \]
\[ \text{ws}(c_1; c_2, G) = \text{ws}(c_1, \text{ws}(c_2, G)) \]

**Strongest Specification-Transformer Semantics (section 2.5):**

\[
\begin{align*}
st(\text{skip}, G) &= G \\
st(\text{abort}, G) &= \emptyset \\
st(x := E, G) &= G \circ R(x := E) \text{ as above} \\
st(c_1; c_2, G) &= st(c_2, st(c_1, G)) \\
st(\text{if } B_1 \rightarrow c_1 [ \ldots [ ] B_n \rightarrow c_n \text{ fi}) &= G \circ (R(B_1) \circ st(c_1, Id) \cup \ldots \cup R(B_n) \circ st(c_n, Id)) \\
st(\text{do } B \rightarrow c \text{ od}, G) &= \neg R(B) \land (\exists G_i) \\
\text{where } G_0 = G, G_{i+1} = st(c, R(B) \land G_i)
\end{align*}
\]

Again, the last three formulae hold only for deterministic commands.

**Connections Between These Semantics:**

Given \( R(c) \) then \( st(c, G) \) can be obtained from formula (13) in section 2.5, while \( ws(c, G) \) can be obtained from formula (8) in section 2.4.

Conversely, for deterministic commands, \( R(c) \) can be retrieved from \( ws(c, G) \) by

\[
s'^{-1} R(c) t \iff s'^{-1} ws(c, \{(s', t)\}) s',
\]

and from \( st(c, G) \) by

\[
R(c) = st(c, Id).
\]
Acknowledgements

The authors are indebted to their colleagues in the Reliability Project, in particular to Brian Randell and Graham Wood.

References


Also in: Springer Lecture Notes in Computer Science, Vol. 16.


