Stanford Verification Group
Report No. 16

Computer Science Department
Report No. STAN-CS-80-789

February 1980

ADA EXCEPTIONS: Specification and Proof Techniques

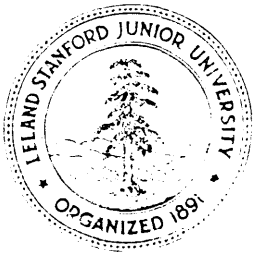
by

D. C. Luckham and W. Polak

Research sponsored by

Advanced Research Projects Agency
and
Rome Air Development Center

COMPUTER SCIENCE DEPARTMENT
Stanford University



Stanford Verification Group
Report No. 16

February 1980

Computer Science Department
Report No. STAN-CS-80-789

ADA EXCEPTIONS: Specification and Proof Techniques

by

D. C. Luckham and W. Polak

ABSTRACT

A method of documenting exception propagation and handling in Ada programs is proposed. Exception propagation declarations are introduced as a new component of Ada specifications. Thi permits documentation of those exceptions that can be propagated by a subprogram. Exception handlers are documented by entry assertions. Axioms and proof rules for Ada exceptions are given. These rules are simple extensions of previous rules for Pascal and define an axiomatic semantics of Ada exceptions. As a result, Ada programs specified according to the method can be analysed by formal proof techniques for consistency with their specifications, even if they employ exception propagation and handling to achieve required results (i.e. non error situations). Example verifications are given.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-80-C-0159 and Rome Air Development Center under Contract F 30602-80-C-0022. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

ADA EXCEPTIONS : SPECIFICATION AND PROOF TECHNIQUES

D. C. Luckham and W. Polak

Abstract: *A method of documenting ezception propagation and handling in Ada programs is proposed. Ezception propagation declarations are introduced as a new component o j Ada specifications. This permits documentation of those exceptions that can be propagated by a subprogram. Ezception handlers are documented by entry assertions. Axioms and proof rules for Ada ezceptions are given. These rules are simple extensions of previous rules for Pascal and define an axiomatic semantica of Ada ezceptions. As a result, Ada programs specified according to the method can be analysed by formal proof techniques for consistency with their specifications, even if they employ ezception propagation and handling to achieve required results (i.e. non error situations). Example verifications are given.*

This research was supported by the Aduanced Research Projects Agency of the Department of Defense under Contract MDA 903 -- 80 -- C -- 0159 and Rome Air Development Center under contract F 30602 -- 80 -- C -- 0022

1. Introduction

Exceptions in the Ada language [I] are intended to provide a facility for dealing with errors and exceptional situations arising during program execution. An exception is an event that causes suspension of normal program execution — called raising an exception. Actions in response to the occurrence of an exception are coded in special units called handlers. When an exception is raised, execution of a handler replaces execution of the current unit. The choice of handlers is dynamic — i.e., determinable only at runtime. Ada exceptions are clearly a powerful programming facility and a study of their semantics and proof is needed.

The obvious semantics of raising an exception is that of a jump to the corresponding handler. In fact, the conventional go to rule [10, 11] is sufficient to describe exceptions with static association of handlers. Problems arise in the case where an exception is not handled locally in a subprogram and the association of the exception with the handler is dynamic. The answer lies in requiring documentation for exceptions that are propagated by procedures or functions; it is then possible to modify the procedure call rule to describe the dynamic association of handlers with exceptions at call time.

We propose a method of specifying exceptions in Ada programs. Axioms and proof rules based on these specifications are given. These define an axiomatic semantics of Ada exceptions, and make possible the verification (proof of consistency with specifications) of Ada programs containing exceptions. Our specifications and proof rules apply to programs with exceptions irrespective of whether exceptions are used only for error situations or as a method of programming normal program behavior (e.g., the program is intended to continue running when exceptions are raised and handled). The proof rules are simple extensions of rules of conventional axiomatics e.g. those of Pascal [7]. Our results show that if the programmer follows a simple specification discipline, then the use of exceptions as a normal programming tool does not greatly complicate the analysis of programs for consistency with specifications.

We restrict presentation of our method to simple cases. In particular, the semantics assumes call by reference implementation for out and in out parameters. Similar proof rules can be given for other implementations of parameter passing. Extensions to more complicated programs, e.g. Ada packages or Ada programs in which exceptions are propagated beyond their scope, are outlined in section 5 and the appendix. We will not deal with the task failure exception propagated between tasks or with suppression of exceptions.

Throughout this paper we will use the notation and terminology of the Ada report [1], except in logical assertions, where we use $a[i]$ to denote the i-th element of array a to improve readability. The word subprogram will be used for a

function or procedure.

A statement of the form **assert** <assertion> is not the assert statement of Ada; <assertion> is a logical assertion and is not intended to be executed. Logical assertions are more general specification statements than Ada conditions, but they obey the normal scope rules: all free variables **are visible program variables.**

2. Specifying Exceptions

The exception facility of Ada includes

— declarations of user **defined** exceptions

— a raise statement that causes an exception, and

— the definition of exception handlers, i.e. code that is to be **executed** whenever a particular exception is raised.

We now **define** the **formal** documentation that has to be provided **for exceptions.**

(1) No documentation is required for exception declarations and raise statements.

(3) For each exception handler the programmer will be required to provide an assertion that is to be true whenever one **of** the **exceptions in** the when clause is raised. Following Ada we will use the syntax:

```
exception-handler ::=
    when exception {| exception }
    assert assertion => sequence_of_statements
```

(3) An exception that is raised in a subprogram and is not handled locally is said to be propagated by this subprogram ([1] 11.34 **We** introduce an exception propagation declaration and require that each exception propagated by **a subprogram** be specified in the subprogram's propagation declaration. Analogous to a handler, a propagation declaration has to provide an assertion that is to be true whenever one of the exceptions in the propagate clause is propagated.

Propagation declarations are placed at the end of the Ada subprogram specification ([1] 6.2). **We** consider propagation declarations as part of a (to be defined) specification language for Ada. As such, they are of the same nature as global, entry, and exit specifications **for** procedures and functions **of** Pascal Plus [11]. **We** use the **syntax:**

propagation_declaration :: =
propagate exception {| **exception** }
assert assertion

subject to the normal scope rules. Note, that a call to a procedure that propagates an exception is a statement that "can raise an exception". **We** restrict such calls to the scope of the exception; the use of propagate declarations **now makes this** restriction compiler checkable. In section 5 we show how these specification requirements may be relaxed by use *of* the others clause. The appendix contains an example of how our specifications will require **modifications** of an Ada program that propagates an exception out of the scope of the declaration of that exception.

In a scope S where an exception *E* can be raised either by a raise statement or a subprogram call the assertion **arrociated** with *E* in this scope is defined as follows:

— If S is a block with a handler for *E* then the **assertion** *of* this handler is the associated assertion,

— *If* S is a block without a handler for *E* then the assertion associated with ***E*** in S is the assertion associated with *E* in the scope immediately enclosing S.

— If S is a subprogram, then the assertion of the propagation declaration for ***E*** is associated with *E*.

Remark: An exception has an associated **assertion** in every scope where **it can be** raised.

For example, consider the following program.

```
procedure p(...)
      propagate E assert A is
begin
      begin
            ...
            raire E;          — B is associated assertion
            ...
      exception
            when E assert B => ....
      end;
      begin
            ...
            raise E;          — A is associated assertion
            ...
      end
end p
```

3. Axiomatic Semantics of Exceptions

3.1. Raising Exceptions

Wherever the statement raise E_i occurs, the above specification principles guarantee that there is an assertion A_i associated with E_i at that point. The semantics of raire E_i is described by the **raise axiom:**

$$\{A_i\} \text{ raise } E_i \{false\}.$$

3.2. Blocks

The rule of exceptions given below requires that any exception handler achieves the exit assertion *of* the block in which it is contained. The handler may assume the truth *of* the assertion in the when clause.

Let C be the block

```
begin
    S0
exception
    when E1 arrert A1 => S1
    when . . .
    . . .
    when En assert An => Sn
end
```

then we have the following rule *of* exceptions:

$$\frac{\{P\}\ S_0\ \{Q\},\ \{A_1\}\ S_1\ \{Q\}\ \ldots\ \{A_n\}\ S_n\ \{Q\}}{\{P\}\ C\ \{Q\}}$$

3.3. Procedure calls

Rules given in 3.1 and 3.2 are sufficient to describe the effect of exceptions handled in the subprogram in which they are raised. We now propose a modified procedure call rule that describes the effect **of** an exception propagated by a procedure. We will restrict our attention to procedures only; a similar treatment for functions is immediate.

Let p be a procedure that propagates the exceptions $E_1 \ldots E_n$ and let $A_1 \ldots A_n$ be the corresponding assertions in the propagate declaration of **p.** Let B_j be the assertion associated with E_j in the calling environment according to the principles in 2. The procedure call rule will require that $A_j \rightarrow B_j$ for all exceptions E_j propagated by p with proper substitutions for in parameters and quantification *for* out and in-out parameters.

The procedure call rule presented below is taken from [9], adapted for **Ada** and exception handling. Let **p** be a procedure with f_i, f_o, and f_{io} being the formal in, out, and in out parameters *of* **p** respectively. We assume that the correctness of the body of **p** has been established with respect to the input condition I(f_i, f_{io}) and output condition $O(f_i, f_o, f_{io})$. Let a_i, a_o, and a_{io} be the corresponding actual parameters of a call to p, then this call is described by the rule:

$$\frac{\begin{array}{l} P \rightarrow I(a_i, a_{io}) \wedge \forall a_o, a_{io}.\big(O(a_i, a_o, a_{io}) \rightarrow Q\big) \\ P \rightarrow I(a_i, a_{io}) \wedge \forall a_o, a_{io}.\big(A_1(a_i, a_o, a_{io}) \rightarrow B_1\big) \\ \quad \ldots \\ P \rightarrow I(a_i, a_{io}) \wedge \forall a_o, a_{io}.\big(A_n(a_i, a_o, a_{io}) \rightarrow B_n\big) \end{array}}{\{P\}\ p(a_i, a_o, a_{io})\ \{Q\}}$$

where the clause $P \rightarrow I(a_i, a_{io}) \wedge \forall a_o, a_{io}.(A_j(a_i, a_o, a_{io}) \rightarrow B_j)$ has to be proven for **each propagated exception** E_j with propagate assertion A_j and B_j associated with E_j in the calling environment. Note that the treatment of exceptions is **similar to that of the normal output condition; raising exceptions may be viewed as just** another exit of ***p.***

The above rule assumes p to be a restricted procedure without any global variables. It may be extended to provide for global variables. Furthermore, we assume an implementation of procedure calls using call by ***reference*** for out and **in out** parameters. Our rules can be adapted for other procedure call mechanisms. Also, other procedure call rules (e.g. [2,5,7]) can be extended to include the conditions $A_i \rightarrow B_i$.

4. Examples

4.1.

Consider **the.** procedure ***seurch*** with in parameters $a, n,$ and x and out parameter i with the **entry** assertion ***true*** and the **exit** assertions $1 \leq i \leq n \wedge x = a[i]$. In case that the value x is not in a, ***seurch*** will raise the exception notfound. For this exception ***search*** contains a propagation with the assertion $\forall j.(1 \leq j \leq n \rightarrow x \neq a[j])$. Altogether we **have** the procedure declaration

type *narray* **is array** (1. .*n*) **of** integer;

procedure *search*(*a*: in *narray*; *n*: **in integer**; *x*: **in integer**; *i*: **out integer**)
    entry ***true***
    exit $1 \leq i \leq n \wedge x = a[i]$
    propagate not found **assert** $\forall j.(1 \leq j \leq n \rightarrow x \neq a[j])$ **is**
begin
    ...
    raire ***notf ound;***
    ...
end

A proof of the body of ***search*** may use the **axiom**

$$\{\forall j.(1 \leq j \leq n \rightarrow x \neq a[j])\} \textbf{ raire } notfound \{false\}$$

Procedure *search* **could be used in the** following context *(m,* ***y,*** and ***b*** are declared in some global scope):

```
assert true;
begin
    search(b, m, y, k);
exception
    when notfound assert ∀j.(1 ≤ j ≤ m → y ≠ b[j]) =>
        k := 0
end;
assert {y = b[k] v (∀j.(1 ≤ j ≤ m → y ≠ b[j]) ∧ k = 0)}
```

Consistency **of** this block with its specifications can be shown by

$$\forall k\Big((1 \le k \le m \wedge y = b[k]) \rightarrow y = b[k] \vee \big(\forall j.(1 \le j \le m \rightarrow y \neq b[j]) \wedge k = 0\big)\Big) \quad (1)$$

$$\forall k(\forall j.(1 \le j \le m \rightarrow y \neq b[j]) \rightarrow \forall j.(1 \le j \le m \rightarrow y \neq b[j])) \quad (2)$$

$$\forall j.(1 \le j \le m \rightarrow y \neq b[j]) \rightarrow y = b[0] \vee \big(\forall j.(1 \le j \le m \rightarrow y \neq b[j]) \wedge 0 = 0\big) \quad (3)$$

Formula (1) proves that the exit assertion of the call to ***search*** implies the exit assertion of the block (note, that ***P*** and ***I*** **of** the procedure call rule are both **true).** This proves consistency if the call to search terminates normally.

Formula (2) proves that the assertion associated with ***notfound*** inside search implies the assertion associated with ***notjound*** in the calling environment (see call rule). This proves that ***notjound*** is propagated by the call to search only in situationa that the handler **"expects".**

Finally, (3) proves that the handler does indeed establish **the exit condition** of the block (i.e. the handler "handles" correctly).

4.2.

The methods presented in this paper are also useful in proving **that an exception** will never be raised. If we have an exception associated with the assertion ***false,*** then the corresponding handler will **never** be executed since its precondition is never satisfied. Consider the example:

```
assert ∃j.1 ≤ j ≤ m ∧ y = b[j];
begin
    search(b, m, y, k);
exception
    when not found arrert false => < empty >
end;
arrert {y = b[k]}
```

The corresponding verification conditions **are** (numbering **as** in previous **example**):

$$\exists j.(1 \le j \le m \wedge y = b[j]) \rightarrow \forall k.((1 \le k \le m \wedge y = b[k]) \rightarrow y = b[k]) \quad (1)$$

$$\exists j.(1 \le j \le m \wedge y = b[j]) \rightarrow \text{Vk.(Vj.(1} \le \text{j} \le \text{m} \rightarrow y \neq b[j]) \rightarrow false) \quad (2)$$

$$false \rightarrow y = b[k] \quad (3)$$

5. Extensions

5.1. Use of the "others" construct

Our method also applies if an exception handler contains the **otherr** clause. If **a** block contains such a handler,

```
exception
    ...
    when otherr arrert A => ...
```

then in this block *A* is the assertion associated with all exceptions that are raised in the block and are not contained in another when clause in the handler. All proof rules and the raise axiom apply unchanged.

The situation is different in propagate declarations. If we allow an others clause in propagate declaration8 then any exception that is not explicitly mentioned in this propagate assertion can potentially be propagated via the otherr clause. Our method can be extended to handle this case if we accept **a** more complicated procedure call rule.

The general principle of the call rule remains the same: for each exception that can potentially be propagated by the called procedure we have to prove that the assertion **specified** *for* this exception in the propagate declaration implies the assertion associated with the exception in the calling environment.

First, note that a scope containing a call to a procedure with an **otherr** clause in the propagation delcaration must define an associated assertion for every exception that may be raised in it; thus in general **its** handler or propagate declaration also **has to have an otherr** clause. Given **the** above situation we can classify all exceptions as follows.

S_o — set of exceptions explicitly named in handlers of propagate declarations for calling environment.

S_i — set of exceptions explicitly named in propagate declaration for called procedure.

B_j — associated assertion for each $E_j \in S_o$.

A_j — assertion specified in propagate declaration for each $E_j \in S_i$.

B_∞ — assertion for **others** clause in handler (or propagate declaration) of calling environment.

A_∞ — assertion for others clause in propagate declaration **of** called procedure.

To prove that a procedure call **is correct we have to verify that:**

a.) $A_j \rightarrow B_j$ for all $E_j \in S_i \bigcap S_o$,

b.) $A_j \rightarrow B_\infty$ for all $E_j \in S_i - S_o$,

c.) $A, \rightarrow B_j$ **for** all $E_j \in S_o - S_i$, and

d.) $A, \rightarrow B_\infty$.

The procedure call rule has to be augmented by these **cases, where** proper substitutions have to be preformed as described **in** section 3.

A better proposal to reduce the number of necessary propagate declaration8 is to introduce a single global exception, error. If a **user** does not want to include each propagated exception with a corresponding **assertion in a propagate** declaration he may include the handler

```
exception
    when others assert A => raise error;
```

in any subprogram and provide a propagate declaration for error. This **conversion has** been discussed in 12.5. of the **Ada rational but was rejected on efficiency considerations.**

5.2. **Application** to Packager

The method described in this paper is **also** applicable to exceptions propagated **from** within modules (or Ada packages). It applies directly to calls of module procedure8 but requires some changes when applied to module bodies. Let us

briefly outline the key idea without going into the detail8 of a particular method *of* specification and **verification** *of* modules,

Let **us assume** that to the using program a module represents an abstract object together with certain operations (module procedures). Some of these may propagate exceptions, ***overflow*** in a stack *for* instance. Part of the visible specification of module procedures are exception propagation declaration8 using abstract specifications. Then the procedure **call** rule *of* section 3.3 applies **to module procedure** calls as well.

For example, the package in [1] 12.3, could be **specified as**

```
generic (size: integer; type elem)
package stuck is
    over flow, under flow: exception;
    procedure push(E: in elem)
        propagate over flow assert f ull(stack);
    procedure pop(E: out elem)
        propagate underflow assert empty(stack);
end stuck
```

This additional documentation is valuable to the programmer even if a formal **verification** is not attempted. The Ada syntax merely states that any of the module procedures may raise any one of the listed exceptions. The propagate declarations give the extra information that, for example, ***pop*** **will never raise the** exception over flow.

The extra problem cornea in **verifying** that exceptions are raised correctly in the stack body. Internal assertions associated with exceptions may refer to local variables of the module body in addition to parameters of procedures. Thus the condition $f\,ull(stack)$ may internally correspond to a condition $length = size$ for some variable *length* local to the module body. Consequently, **for** *overflow* **the** following raise **axiom is** valid within the stack body:

$$\{length = \text{size}\}\ \text{raise over } flow\ \{false\}.$$

A "module declaration rule" has to *enforce* that the visible propagate assertion for an exception is equivalent to the assertion in the body that describes the internal state in which the exception is propagated.

5.3. Predeflned exceptions

It is conceivable that predeflned exceptions can **be** handled with our method. The si**tuation** here is complicated by the fact that predefined **exceptions are**

usually raised implicitly by numerous different statements and expressions. Therefore, all proof rules have to be changed to account for exceptions. For example, the assignment $x \leftarrow y/z$ in a scope where A is the assertion associated with *divide_error* is not described by the standard assignment axiom $\{P\|^{x}_{y/z}\}x \leftarrow y/z\{P\}$ but rather by the rule

$$\frac{Q \wedge z \neq 0 \rightarrow P\|^{x}_{y/z},\ Q \wedge z = 0 \rightarrow A}{\{Q\}\ x \leftarrow y/z\ \{P\}}$$

This rule differs from the standard assignment axiom in having an extra premiss requiring proof that the associated assertion, A, is true when divide-error is raised. Similar extra premisses must be added to the proof rules for many other language constructs leading to an exponential increase in the number of subproblems such as $Q \wedge z = 0 \rightarrow A$. If, however, A is the assertion *true,* then the additional subproblems are trivial. In many cases associating true with an exception handler will be sufficient to verify that a program continues correctly after an exception has been raised.

Since predeflned exceptions frequently correspond to "runtime errors", an alternate approach towards predeflned exceptions is to verify that for certain sets of input values these exceptions will never be raised. Positive results in this direction are reported in [4].

6. Conclusion

We have given an axiomatic semantics for Ada exceptions based on a method of formally specifying exceptions; the method extends to packages in a natural way. The required documentation appears very natural and should not be too demanding for the programmer. The propagation declaration is a helpful specification even in cases where a formal verification is not desired as is shown by the documentation of the stack module in 5.1. The additional information provided by propagate declarations can be used to check for certain semantic restrictions at compile time, e.g. to check that no exception is propagated out of its scope, or that all exceptions that may be raised within a subprogram and not propagated are handled within that subprogram.

The question of how the task failure exception can be treated in a multi task program and whether this can be done consistently with our method has to be investigated in future research.

Acknowledgement

We are greatful for helpful conversations **with Jean Ichbiah and Steve German.**

References

1. ***Preliminary Ado*** **reference manual and** ***rationale*****; Sigplan Notices 14,6** (1979)

2. **Cartwright, R.S.,** Oppen, **D.C.:** ***The Logic of Aliasing*****; Proc. 5th. ACM** Symp. on Principles of Programming Languages, Tucon (1978)

3. **Department of Defense:** ***STEELMAN requirements for high order computer programming languages*****;** (June 1978)

4. **German, S.M.:** ***Automating proofs ojthe absence oj common runtime errors;*** **Proc.** 5th. ACM Symp. on Principles of Programming Languages, Tucon (1978) pp 105-118

5. **Grier, D., Levin, G.:** ***A procedure cull proof rule (with a simple explanation)*****;** Tech. Report 79-379, Cornell University (1979)

6. **von Henke F. W., Luckham D. C.:** ***Automatic Program Verification III: A Methodology for Verifying Programs*****;** Memo AIM-256, Stanford **Artiflcail** Intelligence Laboratory (1974)

7. **Hoare, C. A. R.:** ***Procedures and parameters: an axiomatic*** approach; Lecture Notes in Mathematics 188, Springer (1971)

8. **Hoare,** ***C.*** **A. R.:** ***An Axiomatic Basis of Computer Programming;*** **Comm** ACM **12,10** (October 1969) pp **576-580**

9. **Igararhi, S., London, R. L., Luckham, D.** ***C.: Automatic Program Verification 1: Logical Basis and Its Implementation;*** **Acta Informatica, Vol 4 (1975) pp** 145-182

10. **Knuth, D.E.:** ***Structured Programming with go to etutements;*** Comp. **Surv., 6,4,** (1974)

11. **Stanford Verification** Group: ***Stanford Pascal Verifier, User Manual;*** Report No. STAN-CS-79-731, **Computer** Science **Department, Stanford University (1979)**

Appendix

Propagating cxceptionr out of their scope

The Ada rationale [1] gives the following example of exceptions propagated out of their scope (12.5.2):

```
package D is
    procedure A;
    procedure B;
end

procedure Outside is          — Outside is declared outside the scope
begin                           of error
    ...
    D.A;                      — D.A may propagate error in which case Outside
    ...                         will propagate "something" that cannot be
end                             named syntactically in the declaration of Outside

package body D is
    error: exception;         — scope of error-exception
    procedure A is
    begin
        ... raise error;
    end;

    procedure B is
    begin
        ...
        Outside;              — call to Outside may propagate 'something" which
        ...                     the Ada implementation will recognize as error
    exception
        when error => ...
    end;
end D
```

In this example the scope **of** the exception error is the body of ***D.*** The call ***D.A*** in ***outside*** may propagate error and cause error to be propagated from ***Outside.*** Since ***Outside*** is declared outside the scope of error, this exception cannot be named explicitly in a handler ***for Outside.*** It can be handled only **by**

an others clause. The programmer has essentially lost the ability to distinguish error from other exceptions and to handle it distinctly. In programs, exceptions propagated out of their scope become anonymous. Interestingly, the Ada implementation will maintain the identity of error when it is propagated outside its scope. Therefore, if it is propagated back into its scope again — **as** in this example when ***Outside*** is called from ***B*** — it will be handled by a handler for error and **not by others.** The implementation thus behaves as though all exception propagations **are** within the scope of the exception.

If Ada is extended to require propagate declarations (without an **others** clause), propagation of exceptions beyond their ***scope*** **is** syntactically illegal. For example the package ***D*** must be modified. Since *A* can propagate error and ***A*** is a visible procedure *of* ***D*,** the exception error has to be declared in the visible part of ***D.*** Then the exception, ***D.error*,** is visible to all scopes which **can call** ***D.A.*** Using our documentation, the above example could be written **as:**

```
package D is
      error: exception;              — D.error exception declaration
      procedure A                    — all calls to D.A are within the scope of
            propagate error assert P; D.error
      procedure B;
end

procedure Outside
      propagate D.error assert Q
is
begin
      ...
      D.A;                           — Q is the assertion associated with D.error
      ...                            — a handler for D.error could also be
      ...                              included in outside
end

...
...
```

If one chooses **to** allow **others** clauses in propagate declarations **as** indicated in section 5, then **propagation** of exceptions out of their scope becomes possible. In this **case our method** will work correctly according to **the** Ada **semantics.**

Proof of file transfer example

We consider a slightly modifled version of the file transfer example given **in** the **Ada rationale in 12.3.3.** In addition to propagate and handler assertions, the program is documented with **entry, exit, initial, and invariant** assertions as introduced for Pascal Plus in [11].

We assume that the system routines ***get*** and ***put*** have the following specifications:

```
generic(type t)
    procedure get(j: in out flle of t; x: out t)
    initial j = f0
    entry true
    exit ¬empty(f0) ∧ f0 = append(list(x), j)
    propagate end_of_file assert empty(f0) ∧ j = f0;

generic(type t)
    procedure put( f:in out flle of t; x: in t)
    initial j = f0
    entry true
    exit j = append( f0, list(x));
```

We prove the correctness **of** the following transfer procedure; ***get*** and ***put*** are **assumed** to be appropriate instances **of** the generic procedures declared above:

```
procedure transfer(in j: In out file of t; outj: out file of t)
    initial inj = inf0
    entry empty(out j)
    exit outj = in f0
is
    c:t;
begin
    invariant append(out j, in j) = in f0
    loop
        get(inf, c);
        put(outf, c)
    end loop;
exception
    when end_of_file assert empty(inf) ∧ outj = in f0 =>;
end;
```

The correctness is established by proving the following verification conditions. Note, that instead of introducing universal quantifiers we use new variables to

construct formulas logically equivalent to the quantified formulas in the procedure call rule.

1) Correctnerr of **the handler:** The handler code (empty statement) has to achieve the exit condition of the block:

$$(empty(in\ j) \wedge out\ j = in\ f0 \rightarrow outf = inf0)$$

This verification condition is trivially true.

2) Correctness *of* **raising** the **exception:** The propagate assertion for ***end_of_file*** in ***get*** must imply the handler assertion. Since ***get*** is called within the loop, it may be assumed that the loop invariant is true when a propagation **occurs;** the verification condition has the form

$$invariant \wedge propagate\ assertion \rightarrow handler\ assertion:$$

inf_0 is a new variable denoting the value of the actual parameter in j **of the** call to ***get*** when propagation occurs.

$$(append(out\ j,\ inf) = inf0 \wedge in\ j = inj,0 \wedge empty(in\ j)$$
$$\rightarrow$$
$$empty(inf\_0) \wedge out\ f\ =\ inf0)$$

This verification condition simplifies to true, observing that $append(outj,\ inf) = outf$ whenever $empty(in\ j)$.

3) Correctnerr of entering and leaving the loop: The entry condition of procedure ***transfer*** must imply the loop invariant; when the loop terminates the loop invariant and the negation **of** the loop test $(\neg true)$ must imply the exit condition ***of transfer. outj,0, in j-0*** are new variables denoting the final values of ***outj*** and in j on exit from the loop.

$$\begin{array}{l}(inf = inf0 \wedge \\ empt\ y(out\ j) \\ \quad \rightarrow \\ \quad append(outf, in\ j) = inf0 \wedge \\ \quad (append(outf\_0, in\ f\_0) = inf0 \wedge \\ \quad \neg true \\ \quad\quad \rightarrow \\ \quad\quad outf\_0 = in\ f0))\end{array}$$

Propositional reasoning reduces this condition to

$$(inf = in\ f0 \wedge empty(outf) \rightarrow append(outf, in\ f0) = inf0)$$

which can be seen to be true by properties of append.

4) **Correctness** *of* loop: The loop invariant must be preserved by executing the loop body:

$$
\begin{aligned}
&(append(out\ j,\ in\ j) = in\ f0 \wedge \\
&\neg empty(in\ \mathrm{j}) \wedge \\
&inj = append(list(c\_0),\ in\ f\_1) \wedge \\
&out\ f\_1 = append(out\ j,\ list(c\_0)) \\
&\quad \rightarrow \\
&\quad append(outf\_1, inf\_1) = info)
\end{aligned}
$$

After some substitutions this becomes

$$
\begin{aligned}
&append(out\ j,\ append(list(c\_0),\ in\ \text{j-l})) = in\ f0 \\
&\quad \rightarrow \\
&\quad append(append(out\ j,\ list(c\_0)),\ in\ j\text{-}1) = in\ f0
\end{aligned}
$$

which **is true** by associativity of ***append.***