Industrial Strength Exception Freedom

Peter Amey, Roderick Chapman

Praxis Critical Systems
20, Manvers Street
Bath, BA1 1PX, UK
+44 (0)1225 466991

peter.amey@praxis-cs.co.uk
rod.chapman@praxis-cs.co.uk

ABSTRACT

Ada is unique amongst modern high-level languages in the degree to which it allows programming errors to be trapped at the compilation stage. Using a tool like the SPARK Examiner amplifies this effect and can provide a high degree of confidence that a program is *well formed* before we try and verify that its behaviour is correct. Despite this progress a less tractable class of errors remain: run-time exceptions. For safety-related systems a run-time error may be just as hazardous as any other logical error. For secure systems, guarding against the deliberate generation of such errors—through buffer overflow attacks for example—is vital. The paper explains how automated techniques based on formal verification or proof techniques have now matured and provide an industrial strength solution.

Categories and Subject Descriptors

D2.4 [**Software/Program Verification**] *Correctness proofs, Formal methods, Programming by contract.*

General Terms

Reliability, Security, Languages, Verification.

Keywords

Exception freedom, run-time errors, Ada, SPARK, security, safety, high-integrity systems, DO178B, Common Criteria.

1. INTRODUCTION

Nearly 10 years ago, at the 1993 Ada UK Conference, Program Validation Ltd. presented a paper on the “Automatic Proof of the Absence of Run-time Errors” [1]. Much of the paper was speculative and concerned how the emerging Formal Definition of the SPARK language could form a basis for reasoning about run-time errors.

A great deal has happened in the intervening period: Program Validation Ltd. has become part of Praxis Critical Systems; SPARK[1] has become a well-established language for the development of critical systems; its supporting tools have grown in scope and power; and the inexorable march of Moore’s law has vastly increased the “computing horsepower” available to developers (although much of it remains unused in typical development environments).

The combination of these factors has resulted in the concepts of the 1993 paper maturing into industrial-strength tools, capable of straightforward deployment, and providing the means to eliminate—and *prove the elimination of*—all the predefined exceptions from an Ada program.

2. THE PROBLEM

Over the short history of our industry there has been a trend towards detecting errors earlier in the development lifecycle. As we have migrated from machine code, through assembly languages and low level languages like C towards Algol, Pascal and Ada, our compilers get ever better at indicating the mistakes we have made. Static analysis tools such as the SPARK Examiner [4], the Polyspace Verifier [18], and MERLE [9] can amplify the trend and detect even more potential problems before the expensive testing phase is entered; however, there remains a class of errors which are difficult to detect by purely static means. These are *run-time* errors such as numeric overflow, division by zero and so on.

(As an aside, history might show that this trend peaked with Ada: the current fashion for code generation into ill-defined languages from semantic-free design diagrams might be considered a backward step in this regard, although some domain-specific languages and tools do not suffer in this regard quite so markedly).

For critical systems of any sort it is vital that the software behaves *predictably*. We cannot even begin to answer the question “Does this system meet its specification and fulfil its requirements?” until that precondition is met. A system which can unexpectedly raise a run-time error cannot be said to be predictable.

Furthermore, for a critical, real-time system, the occurrence of a run-time error can be just as damaging as any other error of logic or design. For such systems we need to eliminate all forms of unpredictable and erroneous behaviour before deployment.

[1] **Note**: The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on SPARC™ architecture.

SIGAda’02, December 8–12, 2002, Houston, Texas, USA.
Copyright 2002 ACM 1-58113-611-0/02/0012…$5.00.

3. POTENTIAL SOLUTIONS

So how are we to deal with potential run-time errors in our system? We can either handle them (i.e. deal with them when and where they are detected) or seek to show that they will never occur. The latter approach can be tackled by dynamic (testing) means or static (analysis) methods. In practice, as we shall see, systems may use a combination of all these approaches.

3.1 Handling Exceptions at run-time

Ada provides more assistance in this area than other languages through its system of predefined exceptions. Unfortunately, while the reliable detection and signalling of an error close to the point of occurrence is of enormous benefit, it does not solve all our problems and brings with it some unwelcome baggage.

The specification of recovery actions from unexpected behaviour is notoriously difficult; certainly it is harder than specifying the expected functional behaviour of our system. We are faced with making a safe recovery from:

- an unexpected event (for if it was foreseeable surely we would have guarded against it);
- of an unknown cause (a bug, single event upset, data corruption, malicious intrusion or bad input data?);
- of a system in an unknown state (the value of state variables in the system may depend on whether the subprogram which raised the exception was passed its parameters by reference or by copy); and
- in a very short period of time (e.g. for an unstable fly-by-wire aircraft).

Given these rather severe difficulties it seems that we can only rely on exception handling at two programming extremes:

- At a very local, tactical level, within a subprogram and without allowing propagation. Here we are using exception handlers to replace IF statements, perhaps with some increase in clarity and then again perhaps not.
- At a system-wide, global level where a single, catch-all, *when others* handler might be used to restart a system or switch to a standby unit.

Before we can make use of run-time exception handling, we have to accept a certain amount of overhead in the form of checks that the compiler inserts into our object code. These checks bring two significant drawbacks with them: *performance* and *test coverage* problems.

Code containing compiled-in, run-time checks will inevitably be both larger and slower than code which has the checks suppressed and this may present serious difficulties for some systems.

The difficulty with achieving high levels of test coverage for code in which all checks are enabled is perhaps even more serious. Commonly-used standards such as DO178B [2] and the Common Criteria [3] require high levels of test overage to be demonstrated as part of the certification process. Code inserted by the compiler to detect run-time violations may be difficult or impossible to execute using normal testing techniques. In fact we have a rather strange paradox here: the better written the code and the more free from potential run-time errors it is, the higher the proportion of the check code which will be untestable! DO178B does give us some escape routes here with its concept of *deactivated* code but the effort required remains high.

For all these reasons there is invariably strong project pressure to turn off run-time checking in delivered code thus reducing Ada’s run-time security to a level closer to that of C (although, of course, Ada's standard static checks remain.)

3.2 Eliminating exceptions - dynamic techniques

Attempting to show that a code sample is free from potential run-time errors by testing shares all the difficulties generally associated with testing together with the added complication of trying to identify the test data sets that are most likely to expose such errors. Such data cannot be straightforwardly extracted from requirements as recommended by DO178B nor can we assume that dividing input data into equivalence sets will provide the necessary values since the nature of run-time errors is that they might well be triggered by specific values scattered arbitrarily through such sets.

It is in the search for run-time errors by testing that Ada’s system of predefined exceptions is most helpful. At least we can be sure that if an error occurs during testing it will be detected and we will be given a clear indication of its nature and location. Without the exception mechanism we have to wait until unexpected behaviour is revealed some time after the occurrence of the original error; this is clearly much less effective and much less efficient.

As with all testing, we are ultimately faced with the inconvenience, expressed by Dijkstra, that “testing can demonstrate the presence of bugs, but not their absence”. For the most critical systems we must also accept the uncompromising Bayesian mathematics that no feasible amount of testing can provide assurance in the ultra critical region [12-14]. So it is an unfortunate fact that we cannot eliminate the possibility of run-time errors by testing alone.

In practice, systems are typically tested to meet their functional requirements, up to a level of coverage required by a standard and any run-time errors exposed by this process are dealt with. Systems are not usually tested with run-time error detection as a specific objective.

3.3 Eliminating exceptions - static techniques

Static techniques offer a number of advantages over dynamic testing:

- we can potentially cover the entire vector state of the program; i.e. examine all possible paths and all possible data values;
- we can eliminate run-time errors before entering the expensive test phase thus raising efficiency and reducing cost.

Essentially the trade off between static and dynamic techniques is that the former shows that the program *should* work under all circumstances and the latter that it *does* work under a tiny number of specific circumstances.

Unfortunately the kinds of relatively straightforward static techniques such as data- and information-flow analysis [15] are not powerful enough for the more demanding task of statically detecting potential run-time errors. We need to enlist more powerful techniques such as *abstract interpretation* and *proof*. The former technique was first deployed by DERA Malvern (now QinetiQ) in the MERLE tool [9] and more recently by Polyspace [18]. These tools perform an algebraic evaluation of a program computing possible variable ranges at various points and using that information to deduce the conditions under which a run-time error might occur. Abstract interpretation is extremely computationally intensive and requires a linkable closure of a program to be effective.

The use of proof techniques was suggested in [1] and has now been developed into a practical, industrial-strength tool in the form of the SPARK Examiner. Before considering the SPARK approach in more detail we consider some properties that we require of our program and computing environment before any of these static techniques can be feasible.

3.4 Prerequisites for static elimination of exceptions

We can only apply logic in a logically-sound framework or environment. Essentially this requires:

- A two-valued logic: assertions are True or False but never "Maybe".
- An unambiguous language where the symbols we use can only be interpreted in one way.

A crucial part of meeting the first of these requirements is the elimination of random and invalid values from our system. Demonstration of exception freedom often requires us to show that a variable lies in a particular range; this is rather hard if it contains a random value or if it contains a bit pattern that is not even a valid representation of any value it can legally hold. Such values may come from an external hardware device or code written in another language, for instance. We must therefore work in an environment where we can show that all our data is well-defined and legally represented.

Ambiguity is a more subtle problem but just as detrimental to our goal. Programming languages have a seductive visual similarity to mathematics but in practice allow the construction of programs of uncertain meaning. For example, Section 11.6 of the LRM [19] gives compiler writers substantial freedom to re-order expressions (and even statements). If a static analysis tool makes a different assumption about ordering from that actually employed by the compiler then the analysis may be invalid.

The following piece of erroneous Ada raises a constraint error if the marked expressions are evaluated from right-to-left but safely prints the value 13 if evaluated left-to-right.

```
with Ada.Integer_Text_IO;
use Ada.Integer_Text_IO;
procedure Test1
is
  X, Y, Z, R : Integer;

  function F (X : Integer) return Integer
  is
  begin
    Z := 0;
    return X + 1;
  end F;
begin
  X := 10;
  Y := 20;
  Z := 10;
  R := Y / Z + F (X); -- undefined evaluation order
  Put (R);
end Test1;
```

Similar ambiguities arise from the combination of aliasing with a free choice of parameter passing mechanism. To avoid these traps the tool would have to analyse all possible combinations of re-ordering and parameter passing which would be computationally prohibitive and also perhaps generate a number of false alarms. (Note that the fact that the code above is defined as *erroneous* is of no help to us if we are unable to tell it is erroneous and so predict the unexpected behaviour in advance) .

A more sound approach is to design our language so that these ordering effects cannot occur. For example, effective prohibition of function side-effects removes the ambiguity from the code above. Similarly, effective elimination of aliasing renders parameter passing freedoms harmless.

The combination of a logically-sound language and freedom from invalid or random values is an essential prerequisite for a systematic demonstration of freedom from run-time exceptions.

4. THE SPARK APPROACH

The SPARK approach is derived directly from the considerations of the preceding section. First we created an unambiguous language and by careful attention to the elimination of invalid and random data, created a logically sound environment in which to conduct proof work. An approachable description of the SPARK language can be found in [4] and a more rigorous one at [16]. The important properties of the SPARK language in the current context are:

- freedom from implementation-dependent behaviour (SPARK programs are unaffected by such things as sub-expression evaluation order and differences in parameter passing mechanisms);
- language rules which are 100% machine checkable, using fast and efficient algorithms prior to compilation (so that we know in advance that our program really *is* SPARK and that our proof will be valid);
- effective, system-wide detection of all possible data-flow errors thus ensuring that no random values enter the system; and
- strengthened specifications, through the use of annotations (also commonly known as "design by contract"), allowing efficient analysis to begin before the program is complete (and perhaps before it is even compilable).

A useful consequence of the way the SPARK language was designed is its compatibility with the certifiable, reduced or non-existent run-time systems supplied by various Ada compiler vendors. For example (in alphabetical order), ACT's GNAT Pro High Integrity Edition, Aonix's ObjectAda Real-Time/Raven™, DDCI's SCORE, Green Hills' GMART, and Rational's APEX®/MARK. These systems address other issues concerning system certification and it is useful that SPARK programs can be compiled using them. It is also the case that the reduced run-time support permits only a single "last wish" exception handler making proof of exception freedom especially valuable.

With sound foundations in place we can seek to construct an automatic *proof* of the absence of all run-time errors which is valid for all input data within the computational model.

The predefined exceptions and checks of Ada95 [19] are:

Table 1. Ada95 exceptions and checks

Exception	Source
Constraint_Error	*access check*, *discriminant check*, index check, *length check*, range check, division check, overflow check, *tag check*
Program_Error	*erroneous execution, incorrect order dependence, return not executed in function subprogram, elaboration check, accessibility check*
Storage_Error	*exhaustion of dynamic heap storage*, stack overflow
Tasking_Error	*exceptions raised during intertask communication*

The use of SPARK removes the possibility of many forms of run-time error either because

- the language subset does not include the Ada feature concerned; or
- the additional static semantic rules of SPARK allow the error to be detected before the program is run.

An example of the former is the elimination of Tasking_Error because SPARK currently does not permit tasking. An example of the latter is the Examiner's ability to detect statically whether a subtype indication is compatible with the type mark in a subtype definition. By these means, all the errors in italic in the table above are eliminated.

Leaving aside Storage_Error for now, we are therefore left with index check, range check, division check and overflow check. For these checks, we can generate *proof obligations* or *verification conditions* (VCs) which are equivalent to the run-time checks that the *compiler* would insert. If the VCs can be reduced to "True" then we have a proof that the run-time error can never occur. For example, consider the following code fragment:

```
type T is range -128 .. 128;

procedure Inc(X : in out T)
--# derives X from X;
is
begin
  X := X + 1;
end Inc;
```

On entry to `Inc` we can assume that *T'First ≤ X ≤ T'Last* because if this were not true then a run-time error would already have occurred at the point where `Inc` was called (an obligation to show this will of course be needed at the point of call). In order to show that the single executable statement in `Inc` does not cause a run-time error we need to show that at that point *T'First ≤ (X + 1) ≤ T'Last* is true.

The generation of these checks is fully automated by the SPARK Examiner. Two levels of checking are supplied as standard: the first generates all checks except overflow and the second includes overflow checks. Because we do not have a full formalization of Ada real number arithmetic we do not support checks of real values by default. The Examiner can be configured to produce such checks but in this case the VCs are generated assuming that real numbers behave like true mathematical reals rather than their approximate binary representations. In consequence we cannot claim proof of absence of run-time errors for real numbers: an unprovable VC almost certainly indicates a problem but successful proof may not guarantee exception freedom because an error might arise, for example, from cumulative rounding errors. Traditional numerical analysis is required with such algorithms to determine their stability and error bounds.

The Examiner's output takes the form of a file for each subprogram in the examined code. The files are straightforward text files and contain a VC for each check that has been generated. Each VC takes the form of a number of hypotheses, which may be assumed to be true, and a number of conclusions, which must be shown to be true using the hypotheses and *proof rules* which are also generated automatically by the Examiner to describe type and base type ranges, constant values and so on. VCs are expressed in a simple first order predicate language called the Functional Description Language or FDL. Names are preserved from the original SPARK making the VCS reasonably understandable to the programmer; however, it is important to be clear that VCs are no longer *program code*, they are *mathematical formulae*.

The unsimplified VCs for the code fragment are as follows:

```
For path(s) from start to run-time
check associated with statement of line
13:

procedure_inc_1.
H1:    true .
H2:    x >= t__first .
H3:    x <= t__last .
        ->
C1:    x + 1 >= t__first .
C2:    x + 1 <= t__last .
```

To show that the code is free from any possible exception we need to show that

```
H1 and H2 and H3 → C1 and C2
```

Clearly there are going to be a large number of these VCs. There will be one for every exception check specified in the Ada language which may well amount to more than one per program statement. The approach would therefore be impractical if proofs had to be attempted manually. A second tool, the SPADE Automatic Simplifier, has therefore been developed. Unlike heavyweight theorem provers such as PVS [17], the Simplifier is deliberately limited in scope so that it is fast and is guaranteed not to perform speculative substitutions that leave a formula in a less clear state than it found it. The Simplifier has, however, been tuned over the years to be particularly effective on the kinds of VCs that SPARK run-time checks generate. Very high rates of automatic simplification are achieved on error-free code as indicated in the industrial experience section later in this paper. It is also worth noting that the Simplifier produces a log file listing the substitutions it has made and the rules it has consulted; this make it possible to audit the proof process.

When the Simplifier is applied to our example it leaves:

```
procedure_inc_1.
H1:    x >= - 128 .
H2:    x <= 128 .
       ->
C1:    x <= 127 .
```

Clearly we cannot prove the VC since H1 is not relevant and H2 is not strong enough to establish C1. A potential run-time error therefore exists and the VC gives us a strong clue as to the circumstances under which it will occur: calling Inc with an actual parameter equal to T'Last. Some strategies for dealing with unprovable VCs are discussed later.

In practice, the important thing about this process is the degree of automation achieved. Generation of checks from the Examiner requires only the application of the appropriate command line switch. The Simplifier has a companion "make tool" which finds and simplifies all the VCs generated with a single command. Finally, a summarizer tool provides an overview of the number of VCs that have been generated and which are proved or outstanding.

4.1 Practical Issues 1 - Dealing with input data

One important practical matter that must be addressed here is how potentially unreliable sources of input data are handled. Our computational model assumes that a program obeys the canonical semantics of Ada, but how do we stop a "bad value" (or more correctly an *invalid representation*) from entering our program?

The Ada95 LRM identifies this problem, advising that if an object has an invalid representation

> "It is a bounded error to evaluate the value of such an object. If the error is detected, either Constraint_Error or Program_Error is raised. Otherwise, execution continues using the invalid representation. The rules of the language outside this subclause assume that all objects have valid representations." LRM 13.9.1(9)

A particular problem arises with obtaining values from memory-mapped I/O devices, where the device word-size is larger than the number of bits needed to represent the object being read. For instance:

```
type Warning is
        (None, Advisory, Caution, Error);
for Warning'Size use 8;

Input_Port : Warning;
for Input_Port'Address use ...;
```

then

```
procedure Read_Port (V : out Warning)
--# global in Input_Port;
--# derives V from Input_Port;
is
begin
   V := Input_Port;
end Read_Port;
```

In the assignment to V here, no check is required by the language (since left and right sides of the assignment are exactly the same subtype), but it remains possible that an invalid value might be returned owing to the size of type Warning.

In processing this code, the Examiner recognizes that the variable Input_Port has an address representation clause, and therefore assumes that values read from it may *not* be assumed to be valid. Firstly, the Examiner generates a suitable warning:

```
  11        V := Input_Port;
                 ^
---  Warning :393: External variable
Input_Port may have an invalid
representation.
```

Secondly, the Examiner goes beyond the LRM and generates a VC for the assignment statement of the form:

```
H1:    true .
        ->
C1:    input_port >= warning__first .
C2:    input_port <= warning__last .
```

which *cannot* be proven - giving you a reasonable hint that your program is at risk!

Of course, Ada95 supplies us with the 'Valid attribute for exactly this purpose. The usual idiomatic usage is to read an untrusted value into a temporary variable which is then validated before being returned, thus:

```
procedure Read_Port2 (V : out Warning)
--# global in Input_Port;
--# derives V from Input_Port;
is
   Temp : Warning;
begin
   Temp := Input_Port;
   if Temp'Valid then
      V := Temp;
   else
      V := Error; -- a "safe" value for
                  -- instance, or take
                  -- some other action.
   end if;
end Read_Port2;
```

Now for the first assignment of Temp to Valid, we may assume that Temp'Valid is True, so the VC takes the form:

```
H1:    warning__valid(input_port) .
        ->
C1:    input_port >= warning__first .
C2:    input_port <= warning__last .
```

which *can* be proven, since if a value is Valid then that value must lie within the bounds of its subtype.

4.2 Practical Issues 2 - The "Cosmic Ray" problem

During a presentation of this approach, you can bet good money that someone at the back of the room will pop their hand up and say "Aha! But what about cosmic rays!" This is indeed a good point that warrants some consideration.

Our analytical model is valid with respect to certain assumptions—namely the canonical semantics of SPARK, the trustworthiness of a compiler, and the reliability of the underlying hardware. But what if hardware cannot be assumed to be 100% reliable, or if we face the problem of malicious attack, where the system in question may be tampered with in some way? In some systems, these issues are a real problem. In space-borne applications, "Cosmic rays" (more formally known as *Single Event Upsets*) are a known issue. In the world of smart cards, intrusive and malicious tampering is a well-known and productive attack [5].

Clearly, our proofs of exception freedom are not 100% valid under these scenarios. So should we still bother with the proofs at all? We strongly believe that you should! A common approach is to engineer high-integrity systems such that they are robust in the face of such failures—the use of triple or quadruple redundant systems is common in the aerospace industry for instance. Can we do the same with software? Where we have redundancy in hardware, why not build redundancy in the software so it too is more robust in the face of these problems?

At the most extreme end of the spectrum, we could engineer software using the SPARK approach in the following fashion:

- Prove the program is free from exceptions, and rigorously validate all input data. This would give us strong assurance that the program contains no algorithmic or logic errors that could yield an exception.
- Compile and run the program with "checks on" as an extra level of defence.
- Compile and run the program with additional validity checks enabled to continuously check the validity of all program state. We note the GNAT Pro compiler recently added this functionality specifically to address this kind of application [6].
- Use data representations that are amenable to error detection and correction.
- Run with a final "catch all" exception handler that falls back on some system-level backup or fault-tolerance mechanism.

This style of development offers some interesting advantages. Firstly, it increases the probability that an SEU or other random hardware failure would be promptly detected, rather than allowing the system to "run on" with invalid data. Secondly, the possibility of an algorithmic or logic defect is eliminated by proof, so the question "What's gone wrong?" if an exception is raised is significantly simplified.

4.3 Practical Issues 3 - Storage_Error

One run-time error that the SPARK *proof* model does not directly attack is Storage_Error. High-Integrity systems are typically long-running, and have a fixed amount of RAM in which to run, so "memory leaks" are intolerable.

One of SPARK's design goals is that programs should be amenable to analysis of worst-case execution time and memory usage. Several language features of Ada are prohibited in SPARK since they exhibit an unpredictable execution time, memory usage, or have a large impact on the run-time system. The non-static use of the "&" operator and the ability of an Ada function to return an unconstrained array spring to mind in this context.

SPARK simplifies the problem in three ways:

- SPARK can be compiled with absolutely no use of a "heap" data structure or storage pool. There are no explicit allocators in SPARK, and language features that require implicit use of a heap are also excluded.
- All constraints in SPARK are static. From the compiler's point of view, this means that the activation record (or "stack frame") for a SPARK subprogram is always a fixed size—there is no dynamic component.
- SPARK is non-recursive.

These simplifications reduce static analysis for Storage_Error to a simple analysis of worst-case stack usage for a non-recursive program. While the Examiner does not implement this kind of analysis directly (it is highly compiler- and target-dependent), specialized tools for the SHOLIS project [7] were constructed to perform this task with relative ease. The size of the activation record for each subprogram and the program's call-tree can be directly extracted from the assembler listings produced by a compiler. These data can then be combined to produce worst-case stack usage figures using simple rules [11]. In the SHOLIS project, this static analysis was supported and confirmed by a dynamic "high water mark" test of worst-case stack usage.

5. STRATEGIES FOR DEALING WITH UNSIMPLIFIED VCs

After we have generated our VCs and simplified them we will have:

1. A large proportion that have been proved automatically;
2. a small number that have been simplified but not proved; and,
3. possibly, a very small number that the Simplifier has reduced to False rather than the True we were seeking.

The first group are easy. The code associated with these VCs cannot raise a predefined exception for any data values (as long as our von Neumann machine continues to behave like one!). The last group are also straightforward: they represent code that will *always* generate an exception and which clearly needs attention. This category is actually rather uninteresting since moderately competent programmers usually avoid such gross errors and even the most cursory test program would identify the problem.

It is the middle group that is really interesting. There are three reasons why a VC may not be proved by the Simplifier:

1. It may be too complex for the Simplifier;
2. it may require some system domain knowledge, not included in the source code, to prove it; or
3. it may indicate that a run-time error will occur under specific conditions such as particular values of variables.

A number of techniques can be deployed to deal with these residual VCs. For the first group we might deploy a more powerful proof engine such as the SPADE Proof Checker or document an informal but rigorous argument to show that the VC is indeed true. The second group is also amenable to rigorous argument. For example, a 32 bit counter in an aircraft system that started at 0 and incremented once per second during flight would generate an unprovable VC similar to that for the Inc example; however, it would take approximately 64 years to overflow and we could argue that our system was unlikely to be in service that long let alone fly continuously for that period! These rigorous arguments can be indicated to the summarizer tool which will note which VCs have been cleared by this means.

It is the third group that is most interesting because it is *conditional* run-time errors that are the most dangerous and the hardest to find by testing. Consideration of *why* the VC can't be proved usually quickly reveals the circumstances under which the exception will be raised. We can then set about solving the problem by improving the code in some way. Most straightforwardly, we can employ *defensive programming* to guard against the dangerous condition. For the simple Inc program we could guard the suspect line with `if X < T'Last then` and either refuse to perform the increment if an error would occur or signal the error condition back to the caller.

For environments that seek to eliminate run-time errors without proof, defensive programming is the *only* technique that is available. In effect we must make the system watertight by making every subcomponent watertight. The SPARK proof approach provides another rather useful technique: we can strengthen the *specification* of code, without changing its executable statements, by addition of precondition annotations. For example, our Inc example can be strengthened thus:

```
procedure Inc (X : in out T);
--# derives X from X;
--# pre X < T'Last;
```

The precondition now allows Inc to be proved exception free; however, this is no "free lunch", we now find an obligation to prove the precondition is true everywhere that Inc is called. Often moving proof obligations in this way is very productive because the calling environment may well be rich enough to provide the necessary information to complete the proof whereas the called environment may not.

Usually only a very few iterations are required to generate SPARK source code that can be shown to be free from all run-time errors. Defensive programming is only needed where there is a real risk to be guarded against; where preconditions are introduced our understanding of the code is enhanced; and, throughout the process we have log files and summaries that document the arguments that we have used.

6. SOME PROJECTS

The SPARK approach to exception freedom was first presented nearly ten years ago [1]. It was first deployed on a large project (SHOLIS) in 1995 [7] with reasonable success. Following SHOLIS, several significant improvements to the technology were made, most notably:

- The Examiner's VC Generator was improved to reduce the number and complexity of hypotheses generated for each VC.
- SPARK95 was developed, incorporating modular types, and the 'Valid attribute.
- A facility for modeling volatile state allowed device drivers and other similar low-level code to be completely implemented in SPARK.
- A language annotation has been added to allow the user to specify the predefined base-type from which a signed Integer type is derived. This dramatically improves the Simplifier's ability to discharge VCs arising from Overflow_Check.
- The Simplifier was improved to handle VC forms that arise from common SPARK idioms (for instance, a "for" loop over an enumerated type.)
- The Simplifier was improved to simplify expressions involving quantified predicates—these are important since they are generated in connection with array types.

In addition, the computing power available for theorem proving has increased dramatically in the intervening years. These advances bring exception freedom proof into the area where it can be deployed as a *routine part* of a development process, rather than as a special activity that is only attempted in extreme cases.

For the purposes of this study, we have used three large, real-world programs: the SPARK Examiner (which is, of course, written almost entirely in SPARK), the SHOLIS software [7], and another project—here denoted "Project R". Tests have been carried out to measure

- The execution time of the Examiner (performing static analysis and VC generation) and the Simplifier.
- The "hit rate" of the Simplifier - i.e. the ratio of RTC VCs proven by the simplifier to the total number of RTC VCs.

All tests were carried out using SPARK Examiner 6.1 and Simplifier 2.06—the most recent versions shipping to SPARK users—on a single 1.3GHz AMD Athlon processor running Windows 2000. The results were as follows:

Table 2. Examiner and Simplifier Performance

Test Set	Examiner	SHOLIS	Project R
Executable lines (declarations and statements)	56760	16388	22968
Analysis & VCG time	4 mins 58 secs	4 mins 34 secs	2 mins 2 secs
Simplification time	5 hours 19 mins	8 hours 14 mins	1 hour 48 mins
Total RTC VCs	20833	6741	10963
RTC VCs proven by Simplifier	19127	6088	10017
Hit rate	91.8%	90.3%	91.4%

The hit rate matches our expectations, especially given that the Examiner and Project R contain *only* the mandatory SPARK annotations—there are very few explicit pre-conditions, post-conditions and loop invariants, showing how effective RTC proof can be on an "ordinary" SPARK program.

The performance of the Simplifier is encouraging. RTC proof for programs of this size is within the range of a typical "overnight" regression test and analysis run. We also note that a 1.3GHz processor is now effectively obsolete (i.e. you can't buy 'em any more!)—simplification times could almost certainly be halved again with a more modern processor.

7. NEXT STEPS

Exception freedom proof can now be considered a mature technology—several projects are using it as a routine part of their software development strategy with no direct assistance from Praxis. Work continues to improve the technology on several fronts:

- Language expansion. SPARK continues to grow as a language. A useful subset of tagged types was recently added, for instance, although great care was taken to statically eliminate all possible instances of Tag_Check. The next phase of development will expand SPARK to include the Ravenscar Profile. This introduces some interesting new cases of run-time error that will have to be dealt with:

    - Priority ceiling violation,
    - Executing a potentially blocking operation within a protected operation,
    - More than one task blocking on a single protected entry or suspension object.

    We plan to deal with these entirely statically through the use of additional annotations and analyses performed by the Examiner.

- Parallel Proof. Unlike any other approach to run-time exception freedom that we know of, the VCs generated by the Examiner are entirely independent of one another, so the Simplifier can be applied to many of them simultaneously, limited only by the number of processors available. Simple analysis suggests a near-linear speedup could be achieved—Project R, for instance, took 1 hour 48 minutes hours to simplify on a *single* computer. Given 10 such computers, we could complete the same simplification in approximately 11 minutes plus some overhead for communication. This seems worthwhile—most development projects have between 10 and 100 PCs featuring 1GHz or better processors, most of which do nothing most of the time! While the Extreme Programming community [10] have popularized the regular and pedantic application of regression *testing*, we propose runtime exception proof (or, for that matter, proofs of partial correctness) being similarly used on a regular basis—all you need is CPU cycles, and these are remarkably cheap. We term this new style of verification *Regression Proof.* Initial trials with a prototype parallel "make" tool for the Simplifier have yielded encouraging results. We hope to field this technology on more internal projects, and then with customers, in the near future.

- A final area for further work is to simply improve the Simplifier's "hit rate" by making its proof tactics more powerful. The inexorable march of Moore's law means we can implement ever-more powerful tactics, while maintaining "wall clock" simplification times as they are. This study has already highlighted some areas where the Simplifier can be improved. For a project the size of the Examiner, every 1% improvement in hit-rate results in over 200 VCs that *don't* have to manually inspected—a very worthwhile gain.

8. CONCLUSIONS

Program proof, once the domain of theoretical researchers, has come of age. Using proof-based techniques, static proof of exception freedom is tractable and easy for real developers of real industrial systems *right now!* Many SPARK users are using these techniques without any assistance from Praxis; some have even independently reported their work [8].

The effectiveness of the process, especially the qualitative shift from *finding* some possible run-time errors to *proving their absence*, depends on the logical soundness of the programming language being used and the need to avoid random values entering the system. Efficiency is important because tools must be fast if they are to be used, especially if they are to be used *early enough* to provide real benefit.

Proving the absence of all run-time exceptions is a technique that aligns very closely with the needs of the ComSec community (Common Criteria), the Space Community, and Military/Aerospace system developers (DO-178B). While standards may not explicitly **require** this kind of analysis, it makes very good technical and commercial sense to do it!

9. REFERENCES

[1] Jon Garnsworthy, Ian O'Neill, Barnard Carré. *Automatic Proof of the Absence of Run-Time Errors.* In Ada: Towards Maturity - Proceedings of the 1993 AdaUK conference. IOS Press. ISBN 9051991428.

[2] RTCA-EUROCAE. *Software Considerations in Airborne Systems and Equipment Certification.* DO-178B / ED-12B.

[3] Common Criteria for Information Technology Security Evaluation. ISO Standard 15408. http://csrc.nist.gov/cc

[4] John Barnes. *High Integrity Ada: The SPARK Approach*. Addison Wesley, 1997 (reprinted 2001) ISBN 0201175177. http://www.sparkada.com/

[5] Ross Anderson. *Security Engineering.* Wiley, 2001. ISBN 0471389226.

[6] Robert Dewar, Olivier Hainque, Dirk Craeynest, Philippe Waroquiers. *Exposing Uninitialized Variables: Strengthening and Extending Run-Time Checks in Ada.* Proceedings of Reliable Software Technologies - Ada Europe 2002. Springer-Verlag LNCS 2361. pp. 193-204.

[7] Steve King, Jonathan Hammond, Roderick Chapman, Andy Pryor. *Is Proof More Cost-Effective than Testing?* IEEE Transactions on Software Engineering, Volume 26, Number 8, August 2000.

[8] Darren Foulger, Steve King. *Using the SPARK toolset for Showing the Absence of Run-Time Errors in Safety-Critical Software.* in Reliable Software Technologies - Ada-Europe 2001. Springer-Verlag LNCS 2043. pp. 229-240.

[9] Liz Whiting, Mike Hill. *Safety Analysis of the Hawk In-Flight Monitor.* Presented at the 1999 ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering, Toulouse, France.

[10] Kent Beck. *Extreme Programming Explained.* Addison Wesley. ISBN 0201616416.

[11] Roderick Chapman, Alan Burns, Andy Wellings. *Combining Static Worst-Case Timing Analysis and Program Proof.* Real-Time Systems Journal. Volume 11, pp. 145-171. Kluwer Academic Publishers, 1996.

[12] Butler, Ricky W.; and Finelli, George B.: *The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software*. IEEE Transactions on Software Engineering, vol. 19, no. 1, Jan. 1993, pp 3-12.

[13] Littlewood, Bev; and Strigini, Lorenzo: *Validation of Ultrahigh Dependability for Software-Based Systems*. CACM 36(11): 69-80 (1993)

[14] Littlewood, B: *Limits to evaluation of software dependability*. In Software Reliability and Metrics (Procedings of Seventh Annual CSR Conference, Garmisch-Partenkirchen). N. Fenton and B. Littlewood. Eds. Elsevier, London, pp. 81-110.

[15] Bergeretti and Carré: *Information-flow and data-flow analysis of while-programs*. ACM Transactions on Programming Languages and Systems 1985.

[16] Finnie, Gavin et al: *SPARK95 - The SPADE Ada95 Kernel. Edition 3.0,* July 2002, Praxis Critical Systems

[17] `http://pvs.csl.sri.com/`

[18] `http://www.polyspace.com/`

[19] Taft et. al. *Consolidated Ada Reference Manual: Language and Standard Libraries.* Springer-Verlag Berlin, January 2001. LNCS 2219. ISBN 3540430385.