A Case for Exceptions

Tullio Vardanega
European Space Research and Technology Centre
Keplerlaan 1, 2200 AG Noordwijk, Netherlands
tullio.vardanega@esa.int

Abstract

In this short paper we argue in favour of the value of language support for exceptions. To make our case, we first spell out what we consider exceptions to be about and then discuss reasons for the timid use of them by Ada programmers so far. Finally, we present requirements for an enhanced exception model that, we expect, would increase the confidence and the take up of users.

1 Motivation

Ada 95 has made a major effort at widening the spectrum of application domains that can favourably contemplate it as their implementation language of choice.

While the success of this initiative earns the language a prosperous future, which is to everybody's benefit, it is important that the needs of the niche communities that first pledged allegiance to the language continue to be attended to. One of these niche user communities is that of on-board embedded systems.

In this paper we represent the user view of that particular community in the respect of the (desired) model for exceptions in Ada. In preparation for it, we interviewed users and practitioners in the target user community and processed their contribution into structured arguments.

Overall, we make a case for the use of exceptions in high integrity systems and, concurrently, we put forward the call for some enhancements to the current exception model.

2 Problem Statement

2.1 What We Mean by Exception

For the purpose of this paper, we take exceptions to denote (classes of) erroneous events that occur at the run time of software programs.

We talk of exceptions when the raising of the event and the designated handling of it are promoted synchronously to the program execution, as a direct consequence of it. Conversely, we talk of interrupts and/or of asynchronous transfers of control when we have an arbitrary displacement in time and/or flow of control between the raising of the event and the designated handling of it. These notions mirror the distinction in place, for example, with the standard SPARC architecture [7] between synchronous and asynchronous traps, whereby the former occur as part of the current execution while the latter originate externally to it.

This distinction, which reflects the present semantics of Ada [6], is important in two respects. Firstly, it stops users from placing misguided expectations on the service that the respective mechanisms can render to the design of a program (so that, for example, exceptions are not mistaken for control flow modifiers or as a communication mechanism). Secondly, it encourages users to associate specific contexts of execution to specific classes of events.

Exceptions typically denote the violation of assertions that hold at either language or program level. Language-level assertions reflect properties of the program that originate from the language semantics, which the language run-time is obviously best placed at enforcing. Program-level assertions reflect properties of the program that emanate from application-specific semantics in the user space, which the user may enforce by use of defensive programming techniques.

2.2 Where Do Exceptions Fit

Systems with integrity requirements devote an important proportion of development effort to (attempt to) prevent the occurrence of violation events during operation. Those systems also pay a considerable deal of attention to the definition of system-level reactions to the erroneous events that cannot be prevented or removed. Successive stages of failure mode, effect and criticality analysis (FMECA) typically support this aspect of the development.

The claim that a system is defect-free can rarely be made because (at least) two inherent limitations reduce the exhaustiveness of any fault prevention and removal techniques: (a) the dimension of the effort required to demonstrably achieve freedom from faults; (b) the impossibility of attaining freedom from faults in the face of unavoidable degradation or perturbation of operation of the system infrastructure.

The former limitation may become prohibitive as an effect of the well-known principle of the diminishing return. The latter may become particularly intense in application domains exposed to harsh or hostile environments and/or to unsafe interfaces.

In keeping with the discussion in section 2.1, the origin of a violation event raised in software may be internal to the software program or external to it.

Prevention and removal techniques may be employed to cleanse the program from internal faults. Tolerance and mitigation techniques may be employed to counter failures that may occur outside the program and consequently inject faults inside it.

Language-level and application-level mechanisms for the detection of erroneous situations and the raising of the corresponding exception events provide support for defect removal. The same mechanisms augmented with exception handling capabilities provide support for fault tolerance.

While fault tolerance is, by nature, a system issue, the software implementation language is the vehicle by which system-level fault tolerance concepts or visions translate into (elements of) the system architecture. The implementation of fault tolerance in modern computer systems thus rests, in part, on the quality and power of the support provided by the programming language and its implementation. Consequently, the design of modern software programming languages that want to cater for fault tolerance can hardly do away with exceptions.

In contrast with other contemporary (and competing) languages of the early 80s Ada 83 chose to support the definition, the raising and the handling of both predefined and user-defined exceptions. With respect to our definition in section 2.1, predefined exceptions represent violations of language-level assertions, whereas user-defined exceptions support the preservation and enforcement of program-level assertions.

Ada 95 has slightly augmented the extent of language support in this area, yet without any profound impact on the original model.

2.3 Why Users Underutilise Exceptions

Paradoxically, Ada exceptions have encountered more reluctance than favour in the very environments that were supposed to most benefit from them. We can attribute the paradox to the combined effect of three fundamental reasons.

The first reason stems from the distance that often incurs between the defined power of a language feature and the level and quality of the support provided for it by actual language technology.

This distance may be overlooked by language designers and language advocates. Yet, it can cause serious repercussions on the implementation of the system (vertical repercussions) as well as on the culture of system designers and implementors (horizontal repercussions).

Vertical repercussions are bad enough, but horizontal repercussions may make the notoriety of a programming language much more pervasively than success stories can ever make the fame of it.

Horizontal repercussions deviously consolidate in coding standards (and corporate culture) which curtail the expressive power of the language availed to the programmer in often exceedingly Draconian ways.

In fact, the exception support provided by early embedded Ada technology did not make a tremendously good reputation for the language feature. Users were uncertain about the exact behaviour of the runtime upon detection and raising of exceptions, and feared the excess complexity that seemed to arise from it. Users had a hard time at finding ways to achieve 100% code coverage in the presence of exception handlers. Users were disappointed with the occasionally intolerable memory overhead incurred from systematic use of exception handling in local scopes.

A large proportion of those users now live in a culture that prejudicially reject exceptions. Interestingly, however, prohibiting the use of exceptions alone does not prevent predefined exceptions from being raised while it makes the program irremediably erroneous on their occurrence.

Corollary 1. These considerations propose a distinct requirement on the quality of the language implementation support provided for exception. Aspects of major importance in this respect include: the resource demand (vis-à-vis memory and CPU cycles) of exception support; the efficiency of the raising mechanism and that of the handler identification technique.

Whereas time and lessons learned have allowed the quality of modern Ada technology to improve significantly, the intervening period has not necessarily healed the wounds of users. By all means, however, it should be the language technology to provide metrics that characterise the cost of exceptions rather than the user to determine it.

The second reason for the paradox stems from the haul of concerns caused by the implementation permissions at ∮11.6 ”Exceptions and Optimization” of the Ada 83 Reference Manual [5] with respect to the semantics of programs in the event of exceptions.

The implications of the clause and the possible consequences of them have been the subject of a variety of interpretations, with obvious drawbacks on the track record of a programming language that especially wished to target high integrity system. In fact, this uneasiness was serious enough to repel a considerable component of the user community even further away from the use of exceptions.

Ada 95 [6] has sharpened the clause by prescribing the preservation of the canonical semantics of legal programs even in the face of program reordering and/or optimisation. Nonetheless, implementations are still granted the permission to yield *undefined results* instead of raising exceptions on the failure of language-defined checks.

Needless to say, implementation permissions of this kind serve needs that fundamentally depart from the urge for a rigorous exception model.

Corollary 2. This observation suggests that language implementations that target high integrity systems should support an error detection policy considerably sharper than the standard permits.
Notions like *bounded error* or *undefined result* allow language implementations to negotiate the level of (distributed) overhead that may be tolerable to their target users. Arguably however, high integrity users would rather lean towards unconditional support for exception detection with a known level of overhead.

The third reason for the paradox arises from the negative feedback caused by the design choices of static analysis tools and environments (cf. e.g.: [2]) which exclude exceptions from the supported language subsets.

The designers of those tools and environments argue that the proof of absence of run-time errors is tractable and certainly preferable to the occurrence of the corresponding (predefined) exceptions.

The problem with predefined exceptions resides in the nature of their raising mechanism, which often is obscure to the application programmer and prone to leaving the program in an undefined state. An equal effect of obscurity and undefinedness also stems from uncontrolled exception propagation. Both issues are real and deserve thoughtful attention.

Exceptions are a useful contributor to fault removal and fault tolerance as long as they permit to formulate a diagnosis on the fault and to elaborate a strategy for the treatment of the error.

An exception raising mechanism that does not allow the formation of a diagnosis disrupts the trust that can placed on continued operation of the system and inevitably calls for interruption of (nominal) service. This error treatment in turn is so drastic and high level that it has no use for local exception handling support.

In fact, attenuation techniques exist that help us cope with the problem effectively.

Common attenuation techniques include the obligation for documentation on the runtime behaviour upon predefined exception and the use of coding styles that reduce the scope of the program exposed to the raising of exceptions, thereby permitting to establish a safe recovery state.

Prohibiting the use of exception handling can only follow from either proved or assumed absence of run-time error. Assuming absence of run-time error for realistic programs is a methodological error and should not be pursued. Proving absence of run-time error needs to preserve during operation the validity of the assumptions that permitted the proof, which may be out of reach for several application domains.

Corollary 3. This set of considerations promotes a distinct requirement for documentation of the runtime mechanisms employed to detect and raise predefined exceptions.
In particular, users whose program integrity relies on exceptions most definitely want a model that lets them know, reliably, the state which the raising of predefined exceptions leaves the enclosing block in.

2.4 Do Not Dismiss Exceptions

The three reasons we have given for the exception rejection paradox effectively preclude correct programmer's consideration of the positive contribution of language-level exceptions to program robustness. As a result, programming styles continue to exist that prejudicially prohibit the use of exceptions.

Prohibiting the use of exception handling while contemplating the occurrence of run-time errors (a precaution often required in high integrity systems) calls for the use of alternative defensive programming techniques whose code branches are intrinsically unreachable to proof tools [4], which denounces the inner contradiction of the approach.

In fact, fault tolerance techniques based on exception handling should complement the use of fault avoidance techniques when the latter cannot achieve proof of absence of run-time error or cannot prevent their occurrence upon injection from unsafe interfaces.

Where coding practices or language technology deliberately exclude the use of exception handling, they should also indicate what alternative techniques could/should be used instead and provide evidence that those are sufficient to meet the applicable requirements.

In fact, not all guidance documents speak against exceptions. Reference [10], which analyses Ada language features for their use in high integrity systems, advises against exception propagation in so far as it hinders flow analysis and symbolic analysis, but it allows the use of all other exception features otherwise.

Reference [8] strongly argues in favour of exception support even under the restrictions imposed by the Ravenscar Profile [1]. So does this paper, too.

Ada 95 provides a sufficient, yet initial, basis to reconcile users with the programmatic value of exceptions.

While we may have good conceptual reasons to oppose the prohibition of exceptions, however, we still lack the means to fully meet the corresponding application requirements. The earlier discussion has highlighted a number of desirable enhancements to the current Ada exception model. In the following we elaborate further on the needs that our analysis has captured.

3 The Exception Model We Want

3.1 Predefined Exceptions

A language with strong typing and very precise static and dynamic semantics inherently needs predefined exceptions to enforce those features at run time. Ada supports predefined exceptions and should continue to do so.

Static analysis of the sequential components of the program may help achieve proof of absence of run-time errors. Yet, unsafe interfaces may still exist that can undermine the proof by injecting erroneous values into the system. Extensive use of strong typing along with language-defined checks provides basic support for programmers to build effective protection against such events. In these situations, the use of exceptions perfectly caters for a sharper separation between normal and abnormal processing. The use of the Valid attribute is the obvious alternative for those who do not especially value this separation in the construction of their defensive programming techniques. Those who opt for the exception-based approach (and we among them), however, would require the language implementation to enforce a sharper error detection policy than ARM∮13.9.1(9) allows, and to definitely raise Constraint_Error on the creation of invalid representations (cf. corollary 2).

The notion of Ada.Exceptions.Exception_Occurrence, along with the associated subprogram support, is the vehicle provided by the language to aid the diagnosis of exceptions whose raising is not visible to the programmer.

Whereas the provision of this support moves in the right direction, we find it way too timid and restrictive.

The private nature of the Exception_Occurrence type obscures the underlying data structure and makes it hard and not portable for the user to programmatically extract distinctive error information from it.

The string value returned by Ada.Exceptions.Exception_Information is an adequate vehicle to convey exception information exclusively for those who operate in debug mode. Non-stop systems that have to be resilient to error situations simply cannot benefit from this feature. They also find no consolation in having to resort to marshaling and unmarshaling the information in and out the stream oriented Write attribute of Exception_Occurrence.

Corollary 4. In all of these situations, the user should definitely have the means to control the data attributes (i.e. the parameters) of exception occurrences, well beyond the provision of string-based exception messages.
The ability to attach attributes to a raised exception is the most natural way for users to facilitate fault diagnosis and to tell different occurrences of the same exception.

In Reference [9] we advocated the need for the user to be able to attach the occurrence of synchronous traps (i.e. hardware-detected errors) to the raising of specific exceptions. As those exceptions would be raised below the application interface, they would fall in and, in fact, extend the category of predefined exceptions. In addition to meeting this particular user demand, selectively exposing the predefined exception raising mechanism to the user would also meet the demand for documentation that we have raised in corollary 3.

3.2 User-defined Exceptions

The use of user-defined exceptions for error handling provides a most practical means to preserve separation between nominal and abnormal control flows.

Disciplined encapsulation of scopes that may raise exception within blocks equipped with the corresponding handlers adds to the program structure and to its clarity. This discipline along with the avoidance of uncontrolled propagation warrant timing analysis of raising and handling exceptions (cf. e.g.: [3]).

Whereas the width of the demand for (user-defined) exceptions cannot be doubted, the issue of concern is the extent to which the programmer can actually control the definition of exceptions. As a matter of fact, the language support in this respect is surprisingly modest.

Most notably, exceptions are *not* integrated with the typing model of the language, which is a very unfortunate design choice. Programmers need to use the keyword exception to declare user-defined exceptions. Those declarations, however, give exceptions a very feeble connotation, which has far more drawbacks than assets. For example, we cannot attach specific user-defined exceptions to the definition of specific instances of abstract data types, which precludes access to the very means the language provides for the treatment of violations of assertions in the user space. In much the same way, one cannot use exceptions as generic formal parameters, which excludes exceptions from the instantiation space of the program.

Corollary 5. Not only exceptions should have the same dignity as other language objects but their identity should also feature no less extendability than 'normal' tagged types. The above discussion strongly calls for a better integration of exceptions with the typing model of the language.
If this was the case and exception types were extendable, users would be fully responsible for the amount of overhead incurred in the raising and handling of exceptions. In return for that burden, however, users would be able to determine the structure and the value of the exception attributes, thereby meeting the demand of corollary 4.

3.3 Summary of Requirements

Corollaries 1 to 5 above have drawn aspects of the discussion into the statement of explicit requirements for an enhanced language-level exception model in Ada.

In order to facilitate cross reference for future discussion of the issue, we conclude this paper by listing the major requirements individually.

R1. Better user control of the overhead associated to declaration, raising and handling of exceptions. (From corollary 1.)

R2. More information and guarantee on the state the raising of predefined exceptions leaves the enclosing block in. (From corollary 3.)

R3. Exception parameters that go beyond string or stream based attributes. (From corollary 4.)

R4. Ability to programmatically distinguish, in a portable manner, between different occurrences of the same exception type. (From corollary 4.)

R5. Better integration with the typing model of the language, with exceptions as extendable (composable) types. (From corollary 5.)

R6. Generic parameters to include exception types. (Part of the rationale to corollary 5.)

Note that corollary 2 in section 2.3 does not contribute to this list because it presents no requirement on the language definition but a demand for language implementations targeting high integrity users.

Disclaimer. The views expressed in this paper are those of the author's only and do not necessarily engage those of the European Space Agency.

References

[1] A. Burns. The Ravenscar Profile. *Ada Letters*, XIX(4):49–52, 1999. ACM Press.

[2] B. Carré and J. Garnsworthy. SPARK - An Annotated Ada Subset for Safety-Critical Programming. In *Proceedings TRI-Ada Conference*. ACM Press, 1990.

[3] R. Chapman, A. Burns, and A. Wellings. Worst-case timing analysis of exception handling in Ada. In L. Collingbourne, editor, *Proceedings the Ada UK Conference*, pages 148–164. IOS Press, 1993.

[4] D. Foulger and S. King. Using the SPARK toolset for showing the absence of run-time errors in safety-critical software. In D. Craeynest and A. Strohmeier, editors, *Proceedings 6th International Conference on Reliable Software Technologies — Ada-Europe 2001*, volume 2043 of *Lecture Notes in Computer Science*, pages 229–240. Springer-Verlag, 2001.

[5] ISO. *Ada Reference Manual*. International Standardisation Organisation, Geneva, CH, 1987. ISO/IEC 8652:1987.

[6] ISO. *Ada Reference Manual*. International Standardisation Organisation, Geneva, CH, 1995. ISO/IEC 8652:1995.

[7] SPARC International. *The SPARC Architecture Manual (version 8)*. Prentice Hall, Englewood Cliffs, NJ (USA), 1992. ISBN 0-13-825001-4.

[8] T. Vardanega and G. Caspersen. Using the Ravenscar Profile for Space Applications: the OBOSS Case. In M. González Harbour, editor, *Proceedings 10th International Real-Time Ada Workshop*, volume XXI(1) of *Ada Letters*, pages 96–104. ACM Press, 2001.

[9] T. Vardanega and J. Gaisler. Lessons Learned from the Implementation of On-Board Tolerance to Physical Faults in Ada. *International Journal of Computer Systems Science & Engineering*, 15(1):19–32, 2000. Special Issue on Developing Fault-Tolerant Systems with Ada (A. Romanowsky and A. Wellings, eds.).

[10] B. Wichmann. Programming Languages - Guide for the Use of the Ada Programming Language in High Integrity Systems. Technical Report ISO/IEC TR 15942, International Standardisation Organisation, Geneva, CH, March 2000. (http://www.dkuug.dk/jtc1/sc22/wg9/n359.pdf).