Except for Exception Handling …

Alexander Romanovsky
Department of Computing Science
University of Newcastle upon Tyne
alexander.romanovsky@ncl.ac.uk

Bo Sandén
Colorado Technical University
Colorado Springs, CO
bsanden@acm.org

Exception handling in Ada has a number of well-known problems. It allows for the propagation of unhandled and anonymous exceptions, it is error-prone and it is inappropriate for some language features such as tasking and tagged types. Ada programs with exceptions can be difficult to understand, develop, modify and analyse, and the exception handling features can be misused in a number of ways.

In this paper we introduce the requirements for good exception handling features. We classify the problems with Ada exception handling into two subsets: serious conceptual problems that require an improvement of the language features, and problems attributable to the misuse of the existing features. Problems in the second category can be solved by improving programmers' understanding of the features and ways of using them.

1. Good Exception Handling

Exception handling was introduced as a disciplined and structured way of handling abnormal system events without complicating the normal code and without resorting to the use of "goto". (It could be argued, however, that the exception handling features rely on a structured and restricted form of "goto".) Many researchers regard exception handling as a means for achieving fault tolerance, and we share this view. In this context, exception raising follows error detection, exception handling equates to error recovery, and units of system structuring are units of exception handling and recovery. By means of exception handling, we can incorporate application-specific fault tolerance into a program. Some fault tolerance is necessary in virtually any program since it is impossible to develop fault-free software and the environment in which it operates may produce unexpected events. This is very different from the trivial view of exception handling as a replacement for a debugger or a dumping tool.

Exception handling is usually an important part of any general structuring technique used in system design. It allows system developers to separate the handling of abnormal situations from normal processing, introduces a dynamic separation of the execution of normal code and handlers, and provides two ways of returning control after a component has executed. Exception handling mechanisms should adhere to the system structure and be an integral part of system design. It is beneficial for system development if structuring units are both exception contexts and units of system design.

In complex modern systems often more than half of the application code is devoted to dealing with abnormal system events [7]. Unfortunately, experience shows that a very high ratio of bugs is related to the handling of exceptions. Although this can be partially attributed to a human inability or unwillingness to rigorously and carefully analyse rare and unlikely events, we believe that the main reasons are as follows:

- System developers often misuse exception handling [6]
- Exception handling features are often error prone. (Without looking deeply into the real cause of the Ariane-5 disaster, one can argue that a proper use of the existing exception handling features could have helped to avoid it.)

Clearly there are ways out of this situation. The solution is to provide powerful and flexible exception handling techniques and develop good practices of applying them. From our point of view there are three main requirements for a good exception handling mechanism: its adequacy to the language features, its ability to hide complexity and its safety.

Exception handling mechanisms should *correspond* to the features the language provides, and, more generally, to the concepts used in system development. Two typical examples are object orientation and concurrency. Using procedure-oriented or ADT-oriented exception handling in systems that are otherwise based on classes forces developers to deal with conflicting mental images, complicates design and code and results in many errors. Using sequential exception handling in concurrent systems causes similar problems [21].

System structuring relies on state and behaviour encapsulation for *complexity hiding*. This principle extends to the handling of abnormal behaviour. Each structuring unit should have internal and external exceptions. Internal exceptions are hidden from the outside and should be handled locally if at all possible. External exceptions notify the environment when an exception cannot be handled internally. Exceptions of these two types are used for very different purposes. Mixing them is similar to using global variables or letting the environment access unit-local variables. Each design unit should include a rigorous description of all external exceptions.

Exception handling features should be *safe* and simple to use to avoid any ambiguities and facilitate system development. All possible kinds of misuse should be detected at compile time or by the run-time support. There are several requirements here [20]:

- Each internal exception should have an associated, internal handler

- Internal exceptions cannot be propagated outside the unit
- External exceptions should be explicitly signaled in the unit code
- All external exceptions that the unit can signal should be explicitly declared in the unit interface.

Parts of the research and development community want to reduce the role of exception handing to dealing with the predefined exceptions only. This is a serious mistake because exception handling is a general mechanism that has to be applied in designing any software and its components. Exception handling naturally promotes recursive system design and should be used at each individual system level. When an exception is raised, an attempt is made to handle it at a given level. If this does not succeed it is propagated to the next higher level of the system structure.

Particularly the high integrity and real-time systems communities have general reservations about exception handling [11, 23]. The concerns are that exception handling may add run-time overhead and that it complicates validation and verification (V&V). For example, the rationale for SPARK Ada [3] excludes exception handling on the grounds that "it is easier and more satisfactory to write a program which is exception-free, and prove it to be so, than to prove that the corrective actions performed by exception handlers would be appropriate under all possible circumstances".

We believe that there is enough evidence that exception handling is vital for designing all complex systems. This is first of all because it is impossible to develop fault-free programs and unwise to assume that the environment in which they operate always functions correctly. Second, there are many situations where exceptions can be avoided only at the cost of obscure work-arounds (such as the notorious return codes). Third, exception handling facilitates system design and structuring and makes it easier for developers to understand system behaviour by separating different concerns. (Arguably, this can facilitate V&V as well.) The challenge is to develop more suitable and less expensive exception handling mechanisms and methodologies for their use, as well as techniques for proving and analysing systems with exceptions. A good example of the latter research in the Ada context is [8].

2. Ada Exception Handling

Ada 83 has a number of well-known problems with exceptions [6, 14, 16]. Ada 95 does not really solve them but rather introduces additional problems when it provides new features such as classes and protected types without offering exception handling features that are sophisticated enough to back them. In this section we discuss features that we believe should be improved in the new versions of the language. Section 3 discusses bad practice in using the existing exception handling mechanism. We realise that classifying each problem into one of these two categories is not always easy to justify. Moreover, better exception handling mechanisms can reduce or prevent misuse. But we believe it is important to consider the categories separately because from our point of view the solutions differ.

2.1. Exception Propagation

A number of anomalies are related to exception propagation [16]. They include uncontrolled propagation of unhandled exceptions, unnoticed task completion and propagation of an exception outside the scope in which it is visible as an *anonymous* exception. Ada does not differentiate between internal and external exceptions and even exceptions that are not visible in the containing scope can be propagated. This propagation is implicit and can easily get out of the programmer's control. It is impossible to learn the origin of an exception and its propagation route when you catch it. The ability to propagate anonymous exceptions is identified as one of the main reasons for this. Handling anonymous exceptions is always confusing and dangerous [5].

One solution would be to *with* the package declaring all exceptions that a subprogram can propagate, which would eliminate anonymous exceptions. But even so, the subprogram designer sees no difference between internal and external exceptions, and can still propagate internal ones outside where they will become anonymous. There is no guarantee that each subprogram body will have handlers for all its internal exceptions including the external exceptions of any subprograms it calls. Moreover, it is impossible to identify from which statement (e.g. subprogram call) an exception has been propagated.

Another solution could be to force programmers to declare all external exceptions that procedures can propagate in the package specification. Unfortunately, this approach does not work for subprogram specifications, and gives no guarantee that all external exceptions are declared. Moreover, the link between a particular subprogram and exceptions is lost. For example, with several tagged types declared in the same package, it is impossible to distinguish between the external exceptions of the different types.

2.2. Type System

Ada does not automatically make the base type exceptions visible to the clients of the derived types [16]. Moreover these exceptions are not part of the abstraction represented by the base type as there is no link between the interface exceptions of the parent and the child. Interface exceptions declared in the parent specification are not derived together with the child type.

The problem is exacerbated in the context of tagged types (classes). Since exceptions do not belong to the class, there is no link between object orientation (i.e. inheritance, subclassing) and interface exceptions. The caller cannot know all exceptions that can be signaled just by looking in the specification of the class, the parent class, etc. Actually there is no difference between parent class exceptions and any foreign exception.

Declaring exceptions in the package where a tagged type is declared is no solution. Several tagged types can be declared in the same package, and it is impossible to say which exceptions declared in the package can be propagated by which type. Moreover these exceptions are not visible without with'ing the package.

A serious restriction is that exceptions are not classes or instances of classes. As a result one cannot create derived types that behave as exceptions. It was hoped that tagged types and exception identities could compensate for this [12] but it has not been demonstrated, and it is doubtful that it is possible to a full extent.

Another problem is discussed in [5]: When several objects of the same class are declared and called in the same block, it is impossible to define which call has resulted in signaling a particular exception. Similarly, operations on different types can propagate the same predefined exceptions, which basically contradicts the idea of typing. The solution is to introduce predefined exceptions as classes, in which case each exception will belong to an instance of a type. This makes it possible to identify which call returns a predefined exception and to design different handlers for different calls.

Research results on how to introduce exception handling into object-oriented (OO) languages have been successfully applied in Java and Modula-3. It is unfortunate from our point of view that the introduction of object orientation into Ada was not accompanied by OO exception handling. The combination of existing exception handling and OO programming is unnecessarily error prone.

In an interesting research effort, Ada was extended to allow exception handlers to be introduced at the object (package) level [9]. A number of Ada applications were analysed to demonstrate that object level exception handling is useful. This result is widely cited by proponents of OO exception handling.

2.3. Concurrency

Ada as a concurrent language requires exception handling features that are sophisticated enough to deal with typical problems arising in programming complex concurrent systems. Two features are particularly relevant for concurrent programming:

- An exception propagated outside a rendezvous block in the callee is propagated to both the caller's and the callee's contexts
- An attempt at a rendezvous with a non-existing task raises a predefined exception in the caller.

It is not difficult to see that this is insufficient for programming complex concurrent systems. That exception handling is not really incorporated in the Ada tasking model can cause many problems [16]. A task dies unnoticed if it fails to handle an exception. This can cause deadlock or at least delay the handling of the original abnormal event until another task decides to rendezvous with the completed task and gets an exception of its own. Moreover, when a task is completed because of an exception one cannot learn the reason as this exception is not propagated to any scope. We need new language features that allow exception propagation between tasks as any task can affect the system execution by terminating with an exception, and this exception is not a part of the task interface.

Ada offers no standard way of propagating an exception out of the task body. There are ad hoc solutions, but they are error prone and obviously not part of the language's exception handling model. Moreover there is no standard way except the rendezvous to involve several tasks in the handling of an exception. We would not need such a feature if we could guarantee that errors are contained in the accept blocks and that recovering only one of the tasks is always enough. But there is much evidence that cooperative concurrent systems need cooperative recovery because erroneous information can be smuggled to a number of tasks and because errors can be caused by erroneous patterns of cooperative behaviour [10].

No attempt has been made to develop exception handling features suited to such new Ada concurrency features as protected objects and asynchronous transfer of control (ATC). This is why exceptions cannot serve as triggering events for ATC, and an anonymous exception can be propagated from a protected object through a subprogram or entry call (as for normal subprograms).

3. Bad Practice

In this section we briefly analyse the main ways of misusing Ada exception handling. There is some solid research on the topic [4, 6, 16]. Clearly the problems discussed in sections 2 and 3 are related because sound language features prevent misuse. The challenge is to improve the exception handling mechanism in ways that make misuse more difficult. We realise that there is always need for both good practice and good language but still feel that in the existing research, language problems are confused with bad practice. Unfortunately, it is hardly possible to develop features that are immune to all kinds of misuse, and programmers neither pay enough attention to exception handling nor understand it well. Practical guidelines can improve language usage. They should contain realistic sample programs, typical examples of misuse and common pitfalls. In the following, we discuss some typical forms of misuse that the exception handling features allow.

3.1. When Others

The **when others** choice allows a treatment of an unspecified exception that can only be very general and imprecise [5]. Although **when others** can be used as a firewall, there is no way to learn what the exception was, and the clause complicates system verification [16]. Programmers are recommended to use **when**

others only when they can identify the exception raised [1]. But in practice, **when others** is widely used for catching unknown exceptions (whether anonymous, predefined, raised, or propagated from a called subprogram), when the programmer does not understand what went wrong. This is clearly a dangerous practice. The program should know what exceptions it can handle. System designers should put special effort into analysing all possible exceptions and developing the best ways of handling them. The only useful purpose of **when others** is similar to that of **others** in a case statement: to ensure that all cases have been covered.

Ada 95 has new features that can help provide additional information about an exception caught by an **others** handler. This is not a general solution as it promotes the bad practice of using **when others** in the first place. Moreover, using the exception message or ID complicates the handling and makes the program less reusable. It is often claimed that **when others** is very useful for dealing with many problems of the Ada propagation model (section 2.1). We believe that this is wrong because of the all reasons above.

3.2. Null Handler

A programmer who thinks of exception handling as some annoying pedantry imposed by the language may specify the handling of an exception as **null** and thus ignore it. Many authors seriously doubt the usefulness of such handlers [4, 16]. It is difficult to find examples where an exception should be ignored. Most of the time it is a sign of poor design or poor understanding of the system execution. It is obviously particularly dangerous to use a null handler together with **when others**.

3.3. Improper Handling

Several more examples of misusing Ada exception handling can be found in [4, 16]. Improper handling causes the majority of them:

- Function fall-through when a handler completes without return or raise
- Unset out parameters when the handling is successful
- Propagation of a new exception from the handler thereby masking the original exception
- Inadvertent exception mapping when a handler for an exception, B, first calls a subprogram that propagates its own exception, C, and then re-raises not B but C.

4. Ada-related Research on Exception Handling

4.1. Tools

A tool for static analysis of exception handling is discussed in [22]. It detects all exceptions that can be propagated from a segment of code (including handlers). When used in conjunction with a set of design and coding guidelines, it makes it easier for programmers to identify defects. The defect concept is application specific. It includes code constructs that cause a program to behave incorrectly or make it more difficult to maintain, and constructs that violate a specific set of guidelines. The tool does not analyse the predefined Ada exceptions. Such a tool is primarily needed because the signature of an Ada subprogram does not include the exceptions that it can propagate.

Another tool, called ADAPT, is developed as a part of a systematic approach to implementing fault tolerant Ada programs [2, 18]. It focuses on analysing exception propagation paths and can find unreachable exception handlers, exceptions that are declared but not used, and exceptions propagated to the environment from the main program. Other checks include the following [18]:

- All exceptions propagated beyond a stated boundary are identified to the designer
- Exception propagation distances are reasonable
- Exception name overloading is bounded
- All exceptions are handled
- There are no loops in the exception invocation chain
- The **others** handlers do not inadvertently handle critical exceptions
- Exceptions are not propagated beyond the visibility scope.

A very different approach to detecting potential problems with exception handling is advocated in [14]. The focus is on studying typical patterns of exception-handling misuse and developing a diagnostic tool capable of detecting them. The reported patterns are: propagation of anonymous exceptions, **null** handlers, accessing shared resources without mutual exclusion when handling exceptions in tasks, mistakes in mapping return codes into exceptions in mixed-language implementations, server task termination due to exceptions, and function completion without returning a value or propagating an exception.

Yet another approach is proposed in [5]. The idea is to extend the subprogram's signature with the list of all exceptions that can be propagated from it. This is done using comments of a special type. A pre-processor compiles the program into standard Ada by adding exception specifications when necessary and modifying the structure of the subprogram handler section. In addition, a special external exception (*error_in_P*, where *P* is the name of the subprogram) is automatically introduced into each subprogram signature to be propagated as a replacement for any anonymous exception.

A safety analysis tool, called Exception Analyser [25], is intended for static detection of all possible situations where the Ada predefined exceptions can be raised. The approach is based on the Architectural Neutral Distribution Format and will be used to provide evidence for building safety cases. An important advantage is that it targets the full Ada language rather than a subset.

4.2. Analysing the Experience

It is becoming invaluable to share experience of using Ada exception handling and show realistic examples. Most exception handling examples in Ada books are simplistic and often misleading. They do not show how the features can be used to tackle serious problems in developing large systems but rather demonstrate how they can assist in debugging. As mentioned earlier, a number of papers discuss patterns of Ada exception handling misuse. Unfortunately, few sources discuss how the full power of Ada exception handling can be systematically used. Brief guidelines are given in [1, 23]. Recent research on design patterns can offer very useful solutions to this problem. Bail introduces exception handling into system design and offers a number of patterns for applying existing Ada features in the disciplined development of complex system that handle exceptions [4]. Exception handling is introduced as an issue in all phases of system development. The paper focuses on the following topics that are vital for designing exception handling for a large system:

- Allocation of responsibility for error detection, propagation and handling
- Exception semantics that includes the category of an exception and its granularity
- Global design patterns with two aspects: factorisation of exceptions, and system partitioning with respect to exception propagation
- Local design patterns useful for programming in the small, and in particular for data based, control based, value based and language based approaches.

In the future it will be important to develop patterns addressing problems discussed in sections 2 and 3, as well as patterns applicable during object-oriented design. For example, [13] introduces patterns for combining class level and cooperative exception handling, and a Java class library backing them. Unfortunately there is no such research in an Ada context.

4.3. Advanced Fault Tolerance Techniques

Recently a number of fault tolerance schemes have been developed using standard Ada, all of which rely on a set of programming conventions and on reusable code that is adjusted for a particular application. Researchers working in fault tolerance have long realised that the best way to put their schemes into practice is not to introduce new language constructs but to develop conventions and provide reusable components. Some of these schemes introduce new approaches to handling exceptions in ways that better suit the application characteristics and the structuring technique used. Such schemes include atomic actions and atomic transactions [24], open multithreaded transactions [17], coordinated atomic actions [26] and N-version programming with exception handling [20]. Unfortunately the schemes are error prone and often difficult to use because they are based on the existing Ada mechanisms, which are ill adjusted to concurrent, object-oriented or safe programming.

4.4. Specification Languages

The specification language Anna [19] introduces formal specification into Ada programs to make it easier for programmers to design software prior to implementation and to maintain and explain software. The specification is developed as a set of annotations inserted into the program code as Ada comments. It is symptomatic that the authors introduce exception propagation annotation to be used in specifying subprograms. The idea is to formally specify both the state of the calling environment when an exception is propagated and a condition of the input parameters of a call under which an exception must be propagated.

5. Possible Solutions

One approach would be to develop a number of conventions and methodologies helping programmers to avoid problems and bad practice [2]. The conventions can be backed by tools such as a pre-compiler. A body of sound research on developing different fault tolerance abstractions using standard exception handling falls into this category (see section 4.3). Another idea is to develop design patterns [13] to avoid some of the problems. These directions need additional efforts, particularly in the context of Ada 95.

Another approach is to develop new exception handling mechanisms for the future Ada standard. We realise the complexity of such a task as Ada is a very rich language, which allows different paradigms and abstractions: procedure and procedure library oriented programming, abstract data types, classes, process-oriented and data-oriented concurrent programming, etc. It is difficult to find a single mechanism that fits all language features, which is why Ada uses the simplest possible - procedure-oriented - exception handling. Besides, there is a considerable amount of Ada legacy code, so upward compatibility is vital.

Safety. Many researchers emphasise the importance of introducing the concepts of internal and external exceptions into system development. This relies on explicit declaration of all exceptions that can be propagated from a subprogram. The main rationale for rejecting the association of the name of the possibly propagated exceptions with each procedure declaration in Ada 83 was "that this would require extra runtime code for filtering the propagation of exception" [15]. It is time to reconsider this position not only because computer power is growing but also because safety and predictability are important enough for many applications to offset the "extra runtime code". This decision together with making the rules more restrictive can help with both designing better programs and detecting a number of safety violations at compile time. It may be a good idea to make programmers write handlers for all internal exceptions and to propagate external exceptions explicitly. One could have different keywords for raising internal exceptions and signaling external ones [26].

A partial solution could be to introduce a predefined interface exception *Failure* and transform all anonymous exceptions into it. Another possibility would be to prohibit the **when others** choice and null handlers.

Type system. The fundamental idea of viewing exceptions as entities that can be neither associated with a particular type nor extended seems flawed. There are several ways to make exception handling object oriented. The subprogram signatures can be extended to include the exceptions that the subprogram can propagate. (This will add safety as well.) That way, exceptions can be associated with types to be derived (e.g. with tagged types) and become a part of the type declaration. This change should be supported by a clear set of rules for exception overriding, inheritance, etc.

Another possible approach is to introduce exceptions as classes. This can make it easier to associate exceptions with subprogram signatures. Abstract exception classes can be introduced to help in developing software starting from the earlier phases of the life cycle. Moreover, additional flexibility in choosing the handler can be achieved if exception classes can be extended by introducing additional handlers as methods. Another interesting idea is to allow programmers to attach handlers to packages, abstract data types or classes.

Concurrency. A general feature for exception propagation between tasks is needed in order to allow several tasks to be involved in the handling of an abnormal event. The client-server propagation, which relies on calls, is clearly insufficient for modern complex applications. At present, developers have to rely on ad hoc synchronisation (using nested rendezvous or ATC, for example) to compensate for the lack of such a feature. Similar features would have to be introduced into distributed programming to allow cooperative handling at the partition level. Besides, we need a way to propagate an exception to a known context when a task completes because of an unhandled exception. Several approaches such as the following could be taken to resolve or alleviate the problems:

- Entries can have external exceptions declared in their signatures
- The task specification can include a specification of exceptions to be propagated from a child to the parent task
- Asynchronous raising of exceptions in another task can be introduced
- The ATC features can be extended to allow an exception from another task to be a triggering event.

6. Conclusions

It is our belief that the general solution for the problems discussed lies in the following:

- Improving exception handling support in the language
- Sharing good practice and supporting it by methodologies, patterns, guides, etc.
- Developing powerful validation and verification tools that can deal with programs that handle exceptions.

These three goals are interrelated. For example, simple and safe exception handling makes V&V easier and system behaviour more predictable. This is why it is useful to develop new exception handling mechanisms while keeping good practice and verifiability in mind. Good practice supported by methodologies and patterns of recommended use will always play an important role, but can only help when special attention is paid to developing code that handles abnormal situations.

Acknowledgment. A. Romanovsky has been supported by EPSRC/UK DOTS and EC IST RTD DSoS Projects.

References:

[1] C. Ausnit-Hood. Ada 95 Quality and Style: Guidelines for Professional Programmers. Lecture Notes in Computer Science, vol. 1344, Springer-Verlag. 1997.

[2] P. T. Brennan. Observation on Program-Wide Ada Exception Propagation. Proc. TRI-Ada'93, 189-195, 1993.

[3] J. Barnes. High Integrity Ada. The SPARK Approach. Addison-Wesley, 1997.

[4] W. G. Bail. Exception-handling Design Patterns. In Advances in Computers, v. 49. Academic Press, 191-238, 1999.

[5] M. Bidoit, M.-C. Gaudel, G. Guino. Towards a Systematic and Safe Programming of Exception Handling in Ada. Proc. 3rd Joint Ada Europe/Ada-TEC Conference, The Ada Companion Series, Cambridge University Press, 141-152, 1984.

[6] G. N. Bundy, D.E. Mularz. Error-Prone Exception Handling in Large Ada Systems. Proc. Ada Europe'93, 153-170, 1993.

[7] F. Cristian. Exception Handling and Tolerance of Software Faults. In Lyu, M.R. (Ed.): Software Fault Tolerance. Wiley, 81-108, 1994.

[8] R. Chapman, A. Burns, A. J. Wellings. Worst-Case Timing Analysis of Exception Handing in Ada. Proc. Ada UK Conference, London, UK, 1993.

[9] Q. Cui, J. Gannon, Data-Oriented Exception Handling, IEEE Trans. Soft. Eng., SE-18, 5, 393-401, 1992.

[10] R. H. Campbell, B. Randell. Error Recovery in Asynchronous Systems. IEEE Trans. Soft. Eng., SE-12, 8, 811-826, 1986.

[11] B. Frisberg. Ada in the JAS 39 Gripen Flight Control System. Proc. Ada Europe'98. Lecture Notes in Computer Science, vol. 1411, Springer-Verlag, 289-296, 1998.

[12] M. Gauthier. Exception Handling in Ada-94: Initial Users' Requests and Final Features. Ada Letters, Jan/Feb, XV, 1, 70-82, 1995.

[13] A. Garcia, D. Beder, C. Rubira. An Exception Handling Software Architecture for Developing Fault-Tolerant Software, Proc. 5th International Symposium on High Assurance Systems Engineering, 311-320, 2000.

[14] C. Howell, D. Mularz. Exception Handling in Large Ada Systems. Proc. Washington Ada Symposium, 90-101, 1991.

[15] J. D. Ichbiah, J. B. P. Barnes, J. C. Hiliard, B. Kierg-Brueckner, O. Roubine, B. A. Wichmann. Rationale for the Design of Ada Programming Language. SIGPLAN Notices, 14, 6, part B, 1979.

[16] C. A. Koeitz, J. C. Knight. Anomalies Encountered in Ada Exception Handling. Dept. of Computer Science. University of Virginia, TR-93-11.

[17] J. Kienzle, R. Jimenez Peris, A. Romanovsky, M. Patino Martinez. Transaction Support for Ada. Proc. International Conference on Reliable Software Technologies - Ada-Europe'2001, Leuven, Belgium, May 14-18. 2001. D. Craeynest, A. Strohmeier (Eds.), Lecture Notes in Computer Science, vol. 2043, Springer-Verlag, 2001, 290-304.

[18] J. D. Litke. A Systematic Approach for Implementing Fault Tolerant Software designs in Ada. Proc. TRI-Ada'90, 403-408, 1990.

[19] D. C. Luckham, F. W. von Hanke. An Overview of Anna, a Specification Language for Ada. IEEE Software, 2, 2, 9-22, 1985.

[20] A. Romanovsky. An Exception Handling Framework for N-Version programming in Object Oriented Systems. Proc. 3rd IEEE International Symposium on Object-oriented Real-time Distributed Computing, California, USA, 226-233, 2000.

[21] A. Romanovsky, J. Kienzle. Action-Oriented Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems. In Advances in Exception Handling Techniques. Lecture Notes in Computer Science, vol. 2022, Springer-Verlag, 2001.

[22] C. F. Schaefer, G. N. Bundy. Static Analysis of Exception Handling in Ada. Software - Practice and Experience, 23, 10, 1157-1174, 1993.

[23] B. A. Wichmann et al. Guidance for the use of the Ada programming language in High Integrity Systems. Ada Letters, XVII, 4, July/August, 47-94, 1998.

[24] A. J. Wellings, A. Burns. Implementing Atomic Actions in Ada 95, IEEE Trans. on Soft. Eng., SE-23, 2, 107-123, 1996.

[25] L. Whiting, M. Hill. Safety Analysis of Hawk in Flight Monitor. Software Engineering Notes, 24, 5, 32-38, 1999.

[26] J. Xu, A. Romanovsky, B. Randell. Coordinated Exception Handling in Distributed Object Systems: from Model to System Implementation. Proc. Int. Conference on Distributed Computing Systems, ICDCS-18, Amsterdam, The Netherlands, IEEE CS, May, 12-21, 1998.