# Ada for the Control of Degradation of Service

**Gertrude Levine**
**Fairleigh Dickinson University**
**levine@fdu.edu**

## Abstract

This paper examines degradations of service in terms of constructs of the Ada programming language that are effective for their control and deployment. Several studies of the software lifecycle have focused on taxonomies of defects or errors. Categories of service degradations, however, have only recently been identified. A classification scheme for service degradations clarifies their relationship to defects and errors and their role in the prevention of failure. With the growth of hard real-time systems, wireless networks, and multimedia applications, a comprehensive understanding of service degradations and their usage has become ever more important.

## I. Introduction

Fault tolerance has been integrated into hardware systems for thousands of years (e.g. duplicate roads, wives, and bridges), affording continuous service if a component[1] is damaged or fails. Redundant hardware components enable continuation of service subsequent to unit impairment. A resultant decrease in service quality[2] is called degradation of service. Service degradations can occur in any combination of four modes [12]:

(1) Multiple components of the same functionality are frequently combined to shorten the time required for service delivery. If some of these components are impaired, the remaining functioning units take longer to provide service. If damaged components are replaced, transfer requires a delay and/or the replacement components can be inferior. We call delays that result from component impairment **times degradations**. (We assume that the loss of any component engenders some delay, even if it is replaced by an equal quality unit.)

(2) An impaired system might deliver a lower quality of input and/ or output data, such as lost signals, static, etc. We say that the system then functions in the mode of **input/output degradation**. (Some systems deliberately degrade the quality of input or output during traffic overload in order to mitigate delays.)

(3) Most hardware systems connect components of different functionality in order to supply service. If components that provide non-essential services are impaired, they can be disconnected, enabling the system to continue to function in the mode of **partial services degradation.** (During traffic overload, some systems abandon components whose services are deemed to be non-essential in order to mitigate delays - e.g. Creusa.)

(4) Assume that an optimal or preferred component is not impaired, but communication concerning its state is corrupted; then the component can be downgraded, i.e., its service incorrectly assigned lower priority in a selection process. If a lower quality component is then chosen instead of the available higher quality unit, we say that the system is functioning in the mode of **priority degradation**.

There are differences between hardware and software as well as many similarities. Hardware faults result from physical characteristics, perhaps caused by the manufacturing process, random wear on mechanical parts, or vandalism. Software defects also arise during the manufacturing process; these are called design defects. Aged software systems do not wear out their mechanical parts, but they can degrade in quality due to factors such as the introduction of defects during maintenance; this is called software degradation. Vandalism also occurs in software systems and is sometimes called an exploit. Faults are sometimes activated in hardware components under stress; defects can also be activated in software systems under stress, such as during repeated computations or heavy traffic. Copies of defective software components duplicate the defect; duplicate hardware components have also been manufactured with identical faults. Yet, hardware components impaired by wear can effectively be replaced by an unused copy of the component. Defective software components cannot; instead variants of software components are developed and maintained to replace, when necessary, components that are deemed to be impaired [17]. Generally, mechanisms for software systems are more complex than for hardware systems. Of chief interest to this paper, however, are the similarities between handling hardware and software impairments,

[1]The term "component" denotes the minimal hardware unit of failure, and thus of fault containment and replication for purposes of reliability. We use the term "component" for the minimal software unit of failure as well.

[2]The quality of a component's service is a measure of its conformance to the optimal levels of service specified by the system.

specifically by forestalling failure using the same modes of service degradation.

Active faults or defects cause degradations of service in both hardware and software systems. System specifications tolerate deviations (redundancies) in inputs and outputs, service times, services rendered, and service components where possible and practical. Thus service degradations can be utilized to forestall errors, as long as their deviations are maintained within specified limits. A fault (defect) has been said to be active when it causes an error [1]. This statement is misleading. Defect activation is necessary, but not sufficient, for the occurrence of errors. The activation of a fault (defect) always causes degradation of service, but only sometimes causes an error [12]. (See Figure 1.) Tolerating and controlling degradations of service within specified limits can prevent errors and the failures that result from unresolved errors. Thus service degradations are effective mechanisms for defect tolerance (software fault tolerance [17]), as they are for fault tolerance.

We specify four modes of service degradation:

- Degradation modes of output and input service (e.g., non-optimal data resulting from active defects).
- Degradation modes of service times (such as non-optimal response times caused by active defects).[3]
- Degradation modes of dependencies (such as partial service sets, optimal service sets that are truncated because of active defects). [4]
- Degradation modes of priority selections (e.g., non-optimal component choices caused by active defects).

Software systems specify redundancies in acceptable services. Toleration of deviations from optimal services allows service continuation during "specified" (i.e., non-defective) conditions, as well as in the presence of active defects. For example, the optimal time for the onset of service is when service is requested; the optimal time for service completion is the sum of the optimal starting time plus the minimal time necessary to complete service requirements. Yet delays are common as part of specified service, such as during waits for service to be rendered to higher priority processes. **Degradations of service times**, on the other hand, are caused by active defects, such as priority inversion, deadlock, and DDoS. If delays continue past the maximum specified time, errors result. If errors are not resolved, failures result.

Deviations from optimal values of input and output data are also tolerated. For example, real numbers have an infinite range and precision. Hardware implementations, on the other hand, are approximations that are limited by the number of bits with which values are represented. These deviations do not result from defects, but are designated by specified service. **Degradation of input or output service**, on the other hand**,** is caused by a malfunction, such as a defective computation. If input or output degradations produce values beyond specified deviation limits, errors and possibly failures result.

Deviations from optimal component sets are also tolerated. System components are dependent upon each other for service completion. As part of specified service, however, systems routinely eliminate components that are deemed to hamper important services (e.g., a software manager choosing to curtail non-essential interface elements in order to speed up response time). **Partial service degradation** [1], on the other hand, occurs when an active defect causes the abandonment of a non-essential component (e. g., a corrupted VoIP packet). Resulting degradation is preferable to cascading errors and failures.

Software systems evaluate priorities in an attempt to select optimal service components during dynamically changing conditions. Many selections are based on heuristics, so that deviations from optimal choices are common in specified service (e.g., enhanced FIFO as an approximation of an optimal page replacement strategy). Design diversity allows systems to select between diverse components for required services following the monitoring of system states. When degradation is suspected, an alternate component is substituted to prevent system failure. **Priority degradation**, on the other hand, occurs when priority criteria are corrupted, so that a non-optimal choice is selected even though a better choice is available. For example, consensus reached using malicious information obtained from Byzantine Generals [2] or through an inappropriate algorithm such as simple majority voting [18] typically results in degraded service.

Service degradations are both an opportunity and a challenge for software engineers. They are an opportunity, since the acceptance of deviations in service requirements can prevent errors and failures. They are a

[3] In general, times degradations are caused by the corruption of the number of times per given interval that requests undergo specified mappings [12]. Thus times degradations also involve such metrics as increased bit error rates and decreased bit rates. Other degradations also can be defined per interval.

[4] Partial services are subsets of dependency degradations [12], which also include memory leaks, lost pointers in doubly linked lists, and other defective dependencies.

challenge, because maintaining deviations within specified limits usually relies on heuristics. It is frequently difficult to distinguish between deviations resulting from specified service and those caused by defects, and thus to pinpoint when degradations begin. Typically, deviations are monitored to identify those that threaten to exceed acceptable limits. Parameters are then adjusted to control suspected degradations [8, 18].

The Ada programming language is designed for the development, execution, and maintenance of very reliable systems [11]. Ada is so well developed that its constructs are highly effective for the deployment and constraint of service degradations. Increased reliability can be obtained by permitting appropriate deviations and using Ada to monitor and manage them. We have classified service degradations as deviations resulting from active defects that cause non-optimal service times, non-optimal inputs and/ or outputs, non-optimal dependencies, and non-optimal priority assignments. We now investigate each class in terms of Ada's mechanisms that are appropriate for its control.

## 2. Times degradations

Deviations from optimal service times are tolerated during most specified services. For example, a process in a time-sharing environment is delayed while other processes are serviced in each of their time-slots. A process in a priority scheduling environment is delayed by the execution of higher priority processes. Degradation, on the other hand, involves a service delay caused by an active defect. An active defect that causes an error always results in delay degradation, either due to error resolution, or lacking resolution, due to the infinite delay of failure. Delay degradation, however, need not cause an error if its deviations can remain within specifications. For example, priority inversion (an example of priority degradation) causes delay degradation, since available higher priority processes are delayed while lower priority processes are chosen for service [6, 16]. As long as higher priority processes complete service within specified time limits, however, errors need not result. Times degradations also occur during jitter, infinite blocking, spam, congestion, page thrashing, and distributed denial of service[5], as well as many other examples.

[5] During system overload, when client demands exceed service capacity, delays and other degradations result. We consider the inability of a system to throttle or discard overload to be a defect in the design process.

Ada's support for the control of degradations in service times includes a wide range of mechanisms. Infinite and unbounded waits can be terminated with a timed entry call. For example, if a component is suspended in an infinite wait, an alternate component for that service can be selected following a time-out; times degradation occurs, but the system continues functioning. (Many of us have similarly chosen alternate web sites following calls to unresponsive sites.)

```
-- The wait is too long. We suspect a
-- dead state. Try another component.
select
       Request_Component_Service;
  or
       delay 60.0;
       Request_Alternate_Component;
end select;
```

Ada's mechanisms facilitate detection of times degradation. Ada.Real_Time.Timing_Events [5, 19], for example, contains constructs for generating and monitoring pulses that remain within necessary intervals of precision. Type TIME, the predefined function, CLOCK, the delay statement, and Ada timers are invaluable for periodic monitoring of deviation patterns and suspected times degradations. Type DURATION is useful for defining intervals of acceptable deviations. Ada's priority assignments and priority scheduling protect tasks that threaten to fail if service is not rendered within a restricted delay.

## 3. Input and Output Degradations

Deviations from optimal input and output values are common in specified service. Digital computers were developed for numeric calculations. Yet, hardware representations of integers and real numbers are clearly approximations of infinite values. Specified service tolerates deviations from optimal numeric ranges in order to obtain efficient hardware implementations. An input or output degradation, on the other hand, occurs if a deviation is caused by an active defect, such as a flawed computation. A computation error in one process can lead to the failure of an entire project. For example, a conversion error in Ariane 5 caused the destruction of five hundred million dollars worth of equipment [15]. If deviations remain within specified limits, however, degradations do not result in errors. The IEEE 754-2008 Standard for Floating-Point Arithmetic [7] recognizes that round-off and cancellation errors often result from repeated computations with binary representations of floating point numbers. A careful choice of computation

algorithm [4] is necessary to mitigate such deviations and maintain results within specified limits.

By the 1930s, audio voice inputs were digitized using sampling and quantizing levels. When transmitting digitized voice, the maximum number of signals that can be sent per second is directly proportional to the assigned bandwidth. Telecommunication carriers increased the degree of multiplexing of voice channels by reducing the width of voice bands, leading to less samples, less quantizing levels, and lower voice resolution and quality, as well as the elimination of high frequency signals altogether. The human ear can tolerate such deviations for routine audio communication, but specified service for storage of concert music, for example, requires higher fidelity – i.e., less deviation from the original (optimal) audio values. Digitized video also involves approximations, including lossy compression employed on consecutive frames. Degradations of input or output data, on the other hand, are caused by active defects. For example, when multimedia packets are dropped following increased traffic loads or protocol violations, higher layer protocols do not attempt recovery. Receiving equipment typically interpolates values from adjacent packets for results that may deviate slightly from the lost data; such input degradations are tolerated as less likely to cause failure than delay degradations involved in recovery. Networks monitor packet loss to maintain degradations within acceptable limits. Similarly, networks connecting sensors and weapons continuously modify parameters to control component behavior [8] and maintain input and output degradations within acceptable deviations.

Ada's wide range of type specifications, as well as strong typing constructs, is invaluable for dimensional analysis and detection of input and output degradations. Ada allows programmers to define their own numeric types and subtypes, including range and precision constraints, so that optimal and acceptable values can be clearly documented and verified. Scalar type specifications assist in eliminating defects during implementation and testing [13]. Subset declarations are useful in identifying deviations from optimal values during execution time, which can then be analyzed for possible degradations. Ada's constructs provide strong support for preventing and detecting unacceptable deviations resulting from computations with floating point numbers [14].

Ada mechanisms include several data types to ensure that computations produce values within specified range and precision. A user defined signed integer type specifies a range of acceptable integer values. Attempting to assign an out-of-range value to a variable of such a type, such as occurs during arithmetic underflow or overflow, raises an exception. Ada supports modular types, for which overflow and underflow do not occur, thus assisting in handling computational degradations without the overhead of exception handling. Programmers also define their own floating point types, specifying range and precision, including double (precision), where needed. Definitions of optimal and tolerable limits on range and precision assist in the acceptance of limited degradation and in the detection of increased degradation. The definition of a fixed point type includes a specification for how many digits must be accurately represented and thus implicitly accepts all deviations in values that would be obtained with additional precision. Declarations of fixed point types define a fixed distance between consecutive values, either in powers of two or ten, such as:

```
type Cash_On_Hand is
   delta 0.01 digits 5
      range
         0.00 .. 999.99;
```

C programmers learn that a test for a real value, such as

```
if (x == 0.7)
```

should be approximated with a match that deviates slightly from 0.7, assuming that x was assigned a value during computation. For example, one might test:

```
if (abs(x - 0.7) < 0.0001)
```

Ada recognizes that the hardware representation of most floating point numbers loses precision. Model numbers are defined as those numbers that are representable in binary with no loss in precision, such as 0.0, 0.5, 0.25, 0.75, 0.125. Other floating point numbers are measured by their presence in an interval between the two closest model numbers – explicitly defining an interval of acceptable deviations. Thus a test in Ada for a floating point number within the smallest machine representable interval around the model number 0.75 might be

```
if X in
   Float'Pred(0.75).. Float'Succ(0.75)
```

We can determine the precision that our floating point numbers must satisfy by a type definition so that an exception is raised when a calculation or input causes unacceptable degradation. Alternatively, a subtype definition can be used to divide the defined interval into two parts – one defining the limits on deviation, and a smaller interval to be used for testing and to warn of

potential degradation. If parameters can be adjusted to limit deviations, exceptions can be averted.

```
type X is digits 15
  range
  Float'Pred(0.75).. Float'Succ(0.75);
subtype X_warning_interval is X
  range
    0.75 .. Float'Succ(0.75);
```

If systems trap numeric errors via exceptions, error handling causes times degradation. The exception handler may choose to adjust the computational output, perhaps allowing service to complete within acceptable degradations of non-optimal data outputs.

## 4. Dependency Degradations

If a component fails, perhaps because of a missed hard deadline, and its service is essential to the software system, the entire system must fail. On the other hand if the system can abandon any dependency on the failed component, deeming it inessential, cascading errors and system failure can be avoided. The resulting delivery of only partial services is one form of dependency degradation.

Tasks have been classified by their service times requirements and also according to the effect on the system of the loss of their service [3].

1) A hard deadline task must complete its service within a specified time limit or risk system failure. (We say that such service is essential and tolerates very limited delay deviation before failure results.)
2) A firm deadline task can miss its deadline occasionally, but then the missed service is valueless. The project can continue to execute, but at a "system-level degradation of service." [3] (We say that the system suffers a degradation of partial services following delay degradation; once a delay limit is exceeded, that iteration of the service is abandoned.)
3) A soft deadline task can provide service even if it misses its deadline, but the delay typically degrades system performance. (The delayed task continues to provide service within allowed limits of delay deviations. The delay is a times degradation, resulting from the active defect that caused the task to exceed its soft deadline. Tolerating times degradation is particularly useful in forestalling errors in soft deadline tasks. Neither errors nor failures need occur.)
4) A non-critical task does not have a set deadline. A non-critical task provides non-essential services and it is said that its failure does not affect system performance [13]. (A non-critical task can miss periodic turns at scheduling caused by the arrival of higher priority tasks. Similar events within specified service can lead to tolerated missed services during which no degradation occurs. If, however, a non-critical task fails to perform its service because of an active defect, then a degradation of partial services occurs according to our definitions [12].)

Partial services degradation can be caused by any type(s) of unresolved error. The error affects a non-essential service, causing failure and abandonment of the service's component. Since the system was not dependent on this component's service, however, system-wide failure is averted. The original error, of course, was caused by an active defect [1]. Dependency degradations also can be caused by lost pointers, incorrect file names, or other deficiencies in components that implement dependencies.

Ada has many mechanisms for discarding components of nonessential services in order to protect the rest of the project. As we saw in our section on times degradation, a timed entry call can be useful for abandonment of a component that is suspected of impairment, although the delay upon which it is based is typically a heuristic. Indeed, components can be dropped if there is any delay at all.

```
--If the service is not called within
--the deadline (60.0), the task
--terminates.
select
    Provide_Non_Essential_Service;
or
    delay 60.0;
    terminate;
-- If the terminate clause is not
-- included, only the select component
-- is abandoned.
end select;

-- A request for Non_Essential_Service
-- is abandoned if that service is not
-- available when selected.
select
    Request_Non_Essential_Service;
else
    null;
end select;
```

Conditional statements and other constructs that drop components under evaluated conditions are used in all languages to drop components as part of specified services. Alternatively, if a component is abandoned because of a perceived error (e.g., incorrect file name), degradation of partial services occurs. Ada specifies and documents such choices with exceptional clarity.

Exceptions are abstractions for software errors in Ada. If an error occurs in a non-essential service, the exception handler can abandon its component. If the exception is propagated to non-essential, dependent components, these can be similarly abandoned. Alternatively, handlers can substitute (perhaps lower quality) components to supply the requested services.

## 5. Priority Degradations

Many choices are made during the software lifecycle. Redundant components, frequently of varying quality, prevent failure of a system when the optimal component is impaired. Examples include redundant databases and mirrored servers. Consensus agreements attempt to select the best choice in the presence of active defects. But what if a decision is made according to defective data? Priority degradation occurs when an active defect causes the assignment of highest priority to a component that is inferior to available alternatives. Corrupted priorities are particularly common in the security field, often caused by defective information that is maliciously supplied [2]. Obviously, defective choices can be inadvertent, as occurs in priority inversion.

In section four, we gave an example of a timed entry call in Ada serving as a heuristic for abandoning a requested service, perhaps one whose component exceeded a firm deadline. What if the value of the delay was too small, resulting from a defective computation or input? What if this defect caused the execution of an alternative, lower quality component? Thus, an incorrect value for a delay (or the corruption of a guard in a select statement, or poor coding, etc.) can cause priority degradation:

```
-- 10.0 is too small a value for delay
-- Optimal_Service becomes available
-- at 10.5, which is within the task's
-- time limit.
select
    Request_Optimal_Service;
or
    delay 10.0;
    Request_NonOptimal_Service;
end select;
```

Defective coding can enable the selection of lower quality services even when higher quality services are available. Replacing "or" with "else", inserting guards, or assigning priorities are Ada mechanisms for correcting the following coding defects.

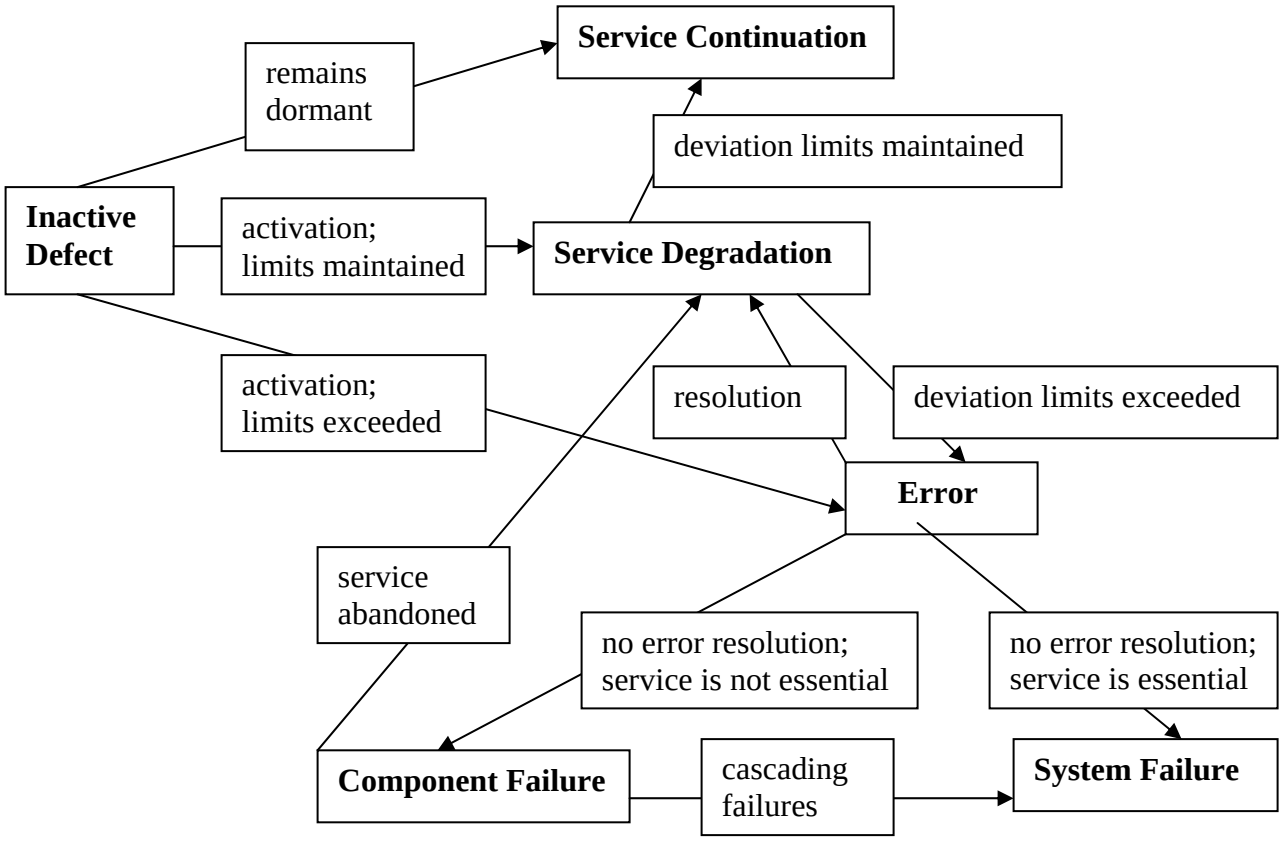
```
-- "or" should be replaced by "else."
-- If Non_Optimal_Service/ Call
-- is selected a priority degradation
-- occurs.

select
     Request_Optimal_Service;
or
     Request_Non_Optimal_Service;
end select;

select
     accept Optimal_Service's_Call;
or
     accept Non_Optimal_Call;
end select;
```

Design diversity is a critical method for defect tolerance [17]. Recovery-block software and multi-version software [9, 10] contain different implementations of the same service, executed sequentially or concurrently, providing redundant components in the event that the preferred implementation threatens to cause failure. Reflection, for which a system monitors its execution and switches between implementations as deemed appropriate, prevents failures by utilizing degradations [17]. Alternate designs are developed, with one, the "primary variant" [17], considered optimal. Service deviations are monitored to ensure that they are maintained within tolerable limits. Parameters are adjusted to limit deviations. If deviations threaten to exceed specified limits, the system chooses a lower quality variant, accepting input/output, times, and/or partial services degradation in order to avoid failure. Using Reflection, an inferior component is chosen, but the decision is obtained through correct information. If, however, a system makes the wrong decision based on corrupted information, priority degradation occurs. The "primary variant" is abandoned even though it would function at a superior level than its replacement.

Measures to achieve design diversity add overhead to a system. If resulting delays exceed time limits, errors will occur. In addition, since these mechanisms are typically complex, they might introduce new defects. Careful study is required to evaluate the cost effectiveness of the prevention of errors with design diversity and the utilization of service degradations [17].

**Figure 1. Defect States and Transitions**

| | Degradation of Service | Error | Failure | Continuation of Service |
|---|---|---|---|---|
| Inactive Defect | activation within specified deviation limits | activation beyond specified deviation limits | n/a | defect remains inactive |
| Degradation of Service | cascading degradations | specified deviation limits exceeded | n/a | deviation limits maintained |
| Error | error resolution | cascading errors | no error resolution | n/a |
| Component Failure | non essential service abandoned | errors in dependent components | cascading failures | n/a |



## 6. Conclusion

Defects cannot always be prevented [17, 18]. There are always costs associated with controlling activated defects. If a defect causes an error, analysis and resolution can be onerous and ineffective. Many errors are avoidable if a system can sufficiently tolerate service degradations following defect activation. A system must analyze whether it can utilize degradations as one of the mechanisms to prevent errors and potential failure. As we have shown, the Ada Programming Language is particularly effective for the design, specification, and monitoring of service degradations.

## References


[1] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy for Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1(1), Jan.-Mar. 2004, pp. 11-33.

[2] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, 2nd edition, Addison-Wesley, 2006.

[3] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems", *Ada Letters*, XXIV (2), June 2004, pp. 1-74.

[4] D. A. Celarier and D.W. Sando, "An Ada Math Library for Real-time Avionics**",** *Ada Letters*, XI (7), Feb. 1991, pp. 274 – 284.

[5] R. M. Clapp, L. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultz, "Toward Real-time Performance Benchmarks for Ada", *CACM*, Volume 29 (8), August 1986, pp. 760-778.

[6] D. Cornhill and L. Sha, "Priority Inversion in Ada," *Ada Letters*, Nov., Doc. 1987, pp. 30-32.

[7] IEEE Standard for Floating-point Arithmetic, *ANSI/IEEE Std* 754-2008, Aug. 29 2008, pp. 1-58.

[8] B. Kallberg and R. Strahle, "Ship System 2000, a Stable Architecture under Continuous Evolution," *Ada Letters*, XXI (4), Dec.2001. pp 47-51.

[9] J. P. J. Kelly, T. I. McVittie, and W. I. Yamamoto, "Implementing Design Diversity to Achieve Fault Tolerance", *IEEE Software*, Vol. 8 (4), July 1991, pp. 61-71.

[10] Y. Kermarrec, L. Nana, and L. Pautet, "Implementing Recovery Blocks in GNAT: a Powerful Fault Tolerance Mechanism and a Transaction Support," *Tri-Ada '95*, Anaheim, California, 1995, pp. 462-266.

[11] J. K. Knight and M.E. Rouleau, "A New Approach to Fault Tolerance in Distributed Ada Programs," *Ada Letters*, VIII (7) June 1988, pp. 123-126.

[12] G .N. Levine, "Defining Defects, Errors, and Service Degradations, *SIGSOFT SEN, Volume 34 (2),* Mar. 2009, pp. 1-14.

[13] J. W. McCormick, "Software Engineering Education: on the Right Track with Ada," *Ada Letters*, XX (3), Sept. 2000, pp. 41–49.

[14] J. W. McCormick, "Ada and Software Engineering Education: One Professor's Experiences," *Ada Letters*, XXVIII, (3), Dec. 2008, pp 91-95.

[15] W. W. Pritchett and J. D. Riley, "An ASIS-Based Static Analysis Tool for High-Integrity Systems," *Ada Letters*, XXVIII (6), Nov, Dec. 1998, pp. 12-17.

[16] J. Ras and A. M. K. Cheng, "Real-time Synchronization on Distributed Architecture with Ada-2005," *Ada Letters*, XXVIII (3), Dec. 2008, pp. 75-84.

[17] P. Rogers, "Software Fault Tolerance, Reflection and the Ada Programming Language, Thesis for the Doctor of Philosophy, University of York, October 24, 2003. http://www.cs.york.ac.uk/ftpdir/reports/2003/YCST/10/YCST-2003-10.pdf

[18] L. Sha, J. B. Goodenough, and B. Pollak, "Simplex Architecture: Meeting the Challenges of Using COTS in High-reliability Systems", *Crosstalk*, Journal of Defense Software Engineering, April 1998, pp. 7-10.

[19] S. Urena, J. Pulido, J. Redondo, and J. Zamorano, "Implementing the New Ada 2005 Real-time Features on a Bare Board Kernel, *Ada Letters*, XXVII (2), August 2007, pp. 61-66.