Aligning Exception Handling with Design-by-Contract in Embedded Real-Time Systems Development

Luis E. Leyva-del-Foyo[1], Pedro Mejia-Alvarez[1], and Dionisio de Niz[2]

luisleyva@acm.org, pmejia@cs.cinvestav.mx, dionisio@iteso.mx
[1]CINVESTAV-IPN, Seccion de Computacion, Av. I.P.N 2508, Col. Zacatenco, 07300 Mexico, DF
[2]ITESO, DESI, Periferico Sur 8585, Tlaquepaque, Jal. 45090, Mexico

Abstract. In this paper we introduce an exception handling mechanism, which is part of the kernel of an operating system for embedded applications. Our approach is based on the theory of design by contract and is adapted for the development of embedded real-time systems.

1 Introduction

The cost of mishandling abnormal situations in general purpose computing (e.g. banking applications, ERPs, text editing, , etc.) can be high. For example, it could mean the improper decrement of the balance of an account upon the failure of an ATM, i.e., it could have decremented the balance and not delivered the money. Exception handling is one of the most important control structures of modern languages (e.g. Java, C++, Eiffel) to reduce this kind of errors. This structure has proven to be very useful in modern systems developed in languages such as Java and Eiffel for general purpose computing.

In embedded systems the cost of mishandled errors can be higher than in general purpose computing. For instance, the incorrect release of the brake of a wheel in an ABS system upon the failure of a sensor could put a car to spin with potentially deadly consequences. Even though embedded systems also benefit from exception handling, the criticality of errors in embedded software has motivated a trend in the use of more thorough techniques to ensure the correctness of embedded software. One of these techniques is the theory of design-by-contract [8]. In this theory, modules interactions are regulated by contracts. A contract defines pre and post conditions and invariants. A module then guarantees that the post conditions and invariants are honored if the preconditions and the same invariants hold when it is called. Even though the current implementation of design by contract include the implementation of exceptions such exceptions has not being made part of the contract. Such an omission diminishes the possibility of the automatic verification of such contracts.

Two additional factors further complicate exception handling in embedded systems. On the one hand, the interaction of the operating system is in general tighter than that of a general purpose application, even to the point of being in the same memory space and linked together (the OS and the application). On the other hand, it is common for an embedded system to interact with the environment in a timely fashion with strict requirements on the reaction time. For instance, in a front airbag system, the computer needs to ensure that it does not take more than 20 ms from the detection of a collision

to the triggering of the chemical reaction that inflates the bag [1]. This type of applications known as real-time applications needs to have a deterministic worst-case execution time. This determinism could be compromised by an exception handling mechanism for general purpose computing. This is due to the triggering of a potentially unbounded list of exception handling code when an exception occurs making the worst-case execution time difficult to measure.
In this paper we describe a novel design of a exception handling structure based on the theory of design by contract that: (a) aligns exceptions to module contracts, (b) enables recasting the interaction with the operating system as a contract, (c) provides constructs to interact with legacy code, (d) is implemented and available for use inside the kernel, and (e) provides mechanisms to bound the exception handling to simplify the measurement of the worst-case execution time of the application.

2 Exception Handling Problems in Embedded Real-Time Systems

As introduced in Section 1, embedded real-time systems can greatly benefit from exception handling. In this section we first discuss the shortcomings related to all types of applications and then those related to embedded real-time systems.

2.1 Exception Handling and Encapsulation Misalignment

Exception handling is a special purpose control structure that enables the separation of the normal code of the software from the abnormal one. By abnormal code we mean code that does not contribute to the behavior the software is required to exhibit but to corrections to deviations from such behavior. For instance, consider the example of a deposit operation to a checking account. The required behavior is to increment the balance of the account by a specified amount. An example deviation that must be controlled is the failure of the disk where the balance is stored. For such case, special code must be added to take care of such situation (e.g., with a retry). The deviating situations are known in programming languages as *exceptions* and the code that handles them as *exception handling code*.
Both, programming control structures for normal code (e.g. *if*s, *while*s, *switch*, etc) and exception handling structures define code blocks around which some control flow is defined. For instance, in an *if-else* structure two code blocks are defined, one is executed if the condition specified in the *if* sentence is true and the other if such a condition is false. Exception handling, on the other hand, defines two types of blocks we identify as: *exception guarded* and *exception handling* blocks. The exception guarded block is where an exception can occur. Such an exception is a condition that a line of code in this block could discover, e.g., a write to disk that failed. Upon the discovery of this condition a sentence to transfer the control to an exception handling block is used. This sentence is known as an *exception raising* sentence. The exception handling code can either finish its execution and continue the execution of the next

sentence after the end of this block or transfer the control to yet another exception handling block or even return to the original exception guarded block.

2.1.1 Encapsulation Corruption

Both control and exception handling structures can be recursively composed. For instance, we can define an *if* block inside a while, switch, or another if block. It is also possible to embed normal control structure blocks inside exception blocks and the other way around. In the end, a fundamental property of both of these types of blocks is encapsulation. By encapsulation we mean two things. On the one hand, that they have a single entry (where execution starts) and exit (where execution continues after it is done with the block) points. On the other hand, that the entry point has a description that enables a programmer to use such a block without knowing its internals. Such a description is both a syntactic description used by the compiler to generate code and a semantic description used by the programmer to understand how to use it[1]. Unfortunately, the encapsulation property gets compromised when normal and exception handling structures are mixed. This is because if a normal control block is encapsulated into an exception guarded block then the *if* control block would have two exit points. For instance, in our deposit example if the code to increment the balance is embedded in an *if* to verify the account number which in turn is embedded in a exception guarded block to correct failed writings to disk, then the if block would have a normal exit point at the end of its block and an exception exit point that transfers the control to the exception handling block. We identify this problem as an *encapsulation corruption* problem.

The encapsulation corruption is, in general, tolerated in control structures. This is because control structure blocks are not opaque, i.e., the detail code of the block is collocated with the code that uses it. In contrast, when modularity structures (procedures and functions) are used, the encapsulation corruption creates important problems. The reason is that the details of the code block is in a different place than the code that uses it (calls it). As a result, the embedding relationship of the modules and the exception structures can be hidden. In particular, determining where the module would return can be difficult.

2.1.2 Encapsulation Corruption in Design-by-Contract

The power of modular encapsulation has been used in a more formal manner in an approach known as design-by-contract [8]. In design-by-contract modules are used along with enforced use contracts. A use contract is composed of three parts: preconditions, invariants and postconditions. A precondition specifies a Boolean expression that must be true before calling the module. An invariant is a Boolean expression that must always be true (before, during, and after the execution of the module). Finally, a postcondition is a Boolean expression the module guarantees to be true after its execution provided that both the preconditions and the invariants were true before its execution. With these contract specifications it is possible to verify the integrations of modules into a system and the properties the system can guarantee.

[1] This semantic description is added as comments, separate documentation and sometime is not present at all.

Given that the design-by-contract methodology relies on the encapsulation to enable sound verification, the corruption of the encapsulation diminishes its utility.

2.2 Exception Handling Shortcomings for Embedded Real-Time Systems

Two shortcomings are specially related to embedded real-time systems: Exceptions across the application and OS boundary and the temporal predictability.

2.2.1 Exceptions Across the Application and OS Boundary

Operating systems provide a set of services to applications that simplifies the computer programming. To ensure the generality of such services (serving the purpose of a variety of applications), options should be given to the applications to customize both their execution and the interpretation of their results. An important part of this interpretation is, hence, the exceptions that can occur during the service execution. For instance, failure to write a file could be treated differently by unattended (e.g. overnight production plan calculations) and interactive (e.g. word processor) applications. In particular, the unattended application may only have the option of aborting the application. In contrast, the interactive application could ask the user for another file path that could include even a different drive. In addition, operating systems need to support multiple languages and hence cannot rely on a single programming language mechanism to handle exceptions. For this reason, it is still a common practice to transform exceptions into return codes that can be easily ignored by the programmer.

Embedded applications have a tighter relation with the operating system. On the one hand, given the limited resource of embedded processors, it is common to link together the application and the OS into a single program. On the other hand, embedded applications are, in general, devices with a single application. This implies that if the application crashes, the whole system crashes (as opposed to a desktop application where a word processor, for instance, could be restarted). This relationship implies that the cost of ignoring exceptions is higher and the consequence of an error in the application can jeopardize the whole system.

2.2.2 Temporal Predictability

Real-time systems must have predictable response times. This implies that their worst-case execution time must be bounded. The state-of-the-practice to find the worst-case execution time is to measure multiple times the execution of the tasks and get the worst measurement. Variations to the execution time are due in part to different paths modules take in the execution of the code. Exceptions can further complicate these paths, potentially leading the execution to a large chain of exception handling blocks that, instead of correcting an exception, can induce timing errors. As a result, it is important to limit the exception handling chaining to avoid creating a timing failure.

2.3 Current Solutions

Different positions have been taken to solve the encapsulation corruption problem related to modularity structures. On the one hand, some languages such as Java [4], enforces *checked exceptions* [5] at compile time. That is, modules (methods in Java) are forced to either capture all exceptions that can be produced by its code or declare the method as a thrower of such exceptions (in the method declaration). Any user of the method would then need to capture the exceptions or recursively declare itself as a thrower of them. Even though checked exceptions prevent ignoring exceptions, not everybody agrees on their benefits. The designers of C# [6] for instance, argue that checked exceptions produces the programming habit of capturing generic exceptions with empty exception handling blocks.
For the design-by-contract framework, Meyer [8] proposes an approach where a module can either finish its execution successfully or with a failure when its contract cannot be satisfied. Such a failure is communicated with an exception. However, the interpretation of exceptions exclusively as failures defies the purpose of the exception structure. That is, the exception handling block cannot serve the purpose of controlling the exception and continuing the execution.
To deal with exceptions within the operating system, some operating systems such as VMS [3], OS/2 [7], and Windows NT [10] have offered support for exceptions. However, no uniform interaction with different programming languages is provided.

3 Design of our Exception Mechanism

In this section we discuss the design of our exception handling mechanism. Our mechanism is designed to be included in the kernel (and was implemented in a kernel) enabling the development of multiple language interfaces. Our initial implementation was developed along with an interface for the C language given its popularity in embedded systems development.

3.1 Basic Design

The base of our exception handling mechanism is composed of an exception guarded block, an exception handling block, an exception raising function, and an exception propagation cancellation function. The construct in the C programming language has the following structure:

```
01:  TRY
02: { /* guarded code */
03:  if (<exception condition>)
04:   RAISE(code, parameter);
05: }
06:  UNLESS
07: {
08:  if (EXCEPTION == code)
09:   { /* exception handling code */
```

```
10:     …
11:     _ABORT(retcode);
12:   }
13: }
14: _END
```

In this construct, the guarded block is identified with a _TRY statement and delimited with braces. This block is followed by an exception handling block that is identified by an _UNLESS statement. Finally, the whole exception block is closed with an _END statement. The _RAISE() function allows a program to raise their own *exception codes* after detecting some abnormal conditions. When an exception is raised inside the _TRY section, the system invocate the code inside _UNLESS section to handle the exception. Then, the program's control flow continues after finishing the _END sentence. Besides this, if the code of the _TRY section is executed without producing any exception, the code of _UNLESS section is left unexecuted, to continue to the code after the _END sentence. The RAISE() sentence includes two arguments, the *code* of the exception and an argument, subject to the interpretation of the applications, that can be a pointer to any complex data type or object.

Contrary to other exception handling blocks, e.g. Java exceptions, the _UNLESS block does not specify the exception it is suppose to handle. Instead the exception code can be checked against the macro EXCEPTION to execute the proper code.
If the exception is properly handled an _ABORT function is used to cancel the exception, meaning that it was successfully handled. In such a case, the execution continues after the _END statement. If, on the other hand, no _ABORT statement is executed the exception is considered unhandled and continues its propagation to other embedding[2] exception handling blocks. We identify this semantics as *propagation-by-default* semantics. It is important to note that in this construct any exception can be identified by an integer code and may have a parameter which can be retrieved with the EXCEPARAM variable.

3.2 Preventing Modular Encapsulation Corruption

To prevent modular encapsulation corruption, we use a twofold scheme. On the one hand, we provide multiple control flow options to terminate the exception handling blocks to support multiple interpretations of exceptions. On the other hand, a mechanism to align the module and exception exit points is also provided.

3.2.1 Enabling Double Exception Semantics

The basic design described in Section 3.1, provides a control flow that facilitates the orderly termination of the guarded block. This semantics delegates the interpretation of the exception to the calling code enhancing the flexibility of the guarded code. However, when the code to be guarded include the full body of a module there could

[2] An exception guarded block can be embedded into other exception guarded blocks recursively as discussed in Section 2.

be exceptions that need to be resolved within the module while others would need to be propagated outside it. To facilitate the resolution of exceptions inside a module our design also provides retry semantics. The retry semantics is build by calling the _RETRY function with a retry code once the exception has been handled in the _UNLESS block. This statement transfers the control back to the beginning of the _TRY block. An example of the retry semantics follows.

```
01: _TRY
02: { /* guarded code */
03:  …
04:  if (_RETRYCODE)
05:   /* Code for retry */
06:  if (<exception condition>)
07:   RAISE(code, parameter);
08: }
09: _UNLESS
10: {
11:  if (EXCEPTION == code)
12:  { /* exception handling code */
13:    …
14:    _RETRY(1);
15:   }
16: }
17: _END
```

In this example, the _UNLESS block has a _RETRY statement with a retry code equal to one in line 14. This _RETRY sentence transfers the control back to the start of the block in line 2 and the retry code is checked with the variable _RETRYCODE to perform special code for the retry. The support for the double semantics – orderly failure and retry – promotes the separation of exception handling code internal to a module from the code that must be delegated to the module's caller.

3.2.2 Aligning Exception and Module Exit Points

The alignment of the exception and module exit points is handled by creating a code block that is always executed whether or not an exception occurs. This block is known as the _FINALLY block. This block is intended to encapsulate the finalization code of a module ensuring that such code will be executed when the control is transferred outside the module. In other words, the _FINALLY block represents a single exit point for both the module and the exception handling block**,** given that both exit paths, the normal termination and the exception termination executes it. An example of such construct follows:

```
01: _TRY
02: {
03:  /* guarded code */
04:  …
05:  if (_RETRYCODE)
06:   /* Code for retry */
07:  if (<exception condition>)
08:   RAISE(code, parameter);
09: }
10: _UNLESS
```

```
11: {
12:  if (EXCEPTION == code)
13:   {
14:    /* exception handling code */
15:    …
16:   }
17: }
18: _FINALLY
19: {
20:  /* Finalization code: free resource, etc.*/
21: }
22: _END
```

In this example, whether an exception occurs or not, the _FINALLY block from line 19 to 21 executes. In our design, the _FINALLY block must include the cleanup actions that must be executed regardless of the occurrence of exceptions. We do not keep a record of the locks retained by any section of code, and in consequence, the programmer is responsible for the release of any locks within the _FINALLY block.

Combining _FINALLY and _RETRY.

The _FINALLY block not only is executed with the orderly-failure semantics of exceptions but also with the retry semantics. This implies that every time we transfer the control back to the _TRY block with the _RETRY statement the _FINALLY block will be executed. This control flow is designed to keep a transactional model of the block where resources (e.g. locks) are acquired at the beginning of the _TRY block and release in the _FINALLY block. As a result, when using the _RETRY statement the resource would be released in the _FINALLY block and reacquired in the _TRY block.

3.3 Exceptions across the application and OS boundary

Given that our exception mechanism is implemented in the kernel, it can be used in the kernel itself. The propagation of an exception that occurs in the kernel toward the application depends on how the OS and the application are related. In embedded systems this relationship can be either tight or loose. By tight we mean that both the OS and the application are linked in the same memory space as a single program (e.g. μC/OS-II and OSEK/VDX). On the other hand, by loose we mean that the OS and the applications are linked as separate programs and the OS loads the application programs (the common model in desktop computers and larger systems).

When the OS and the application are tightly related, the exception handling mechanism can be used without modification and can be used even with contracts. However for legacy applications expecting return codes, the exceptions occurring in the OS must be translated into such codes. Our design includes a _TRYERROR block to wrap system calls to translate exception into return codes. This block is used as follows:

```
01:  int osService() /*Internal code of a system call*/
02: {
03:       _TRYERROR {
04:       osServiceCode();
```

```
05:         }
06:         _END
07:         return;
08:  }
```

A layer of wrappers of this form is then used to support legacy applications that cannot handle exceptions. When the applications and the OS are loosely related, exceptions are not used. Instead such exceptions are translated into error codes with the _TRYERROR block. On the other hand, the termination of applications that uses our exception handling mechanism has to ensure that no exception is left unhandled. For this purpose a *default handler* is included in every task in the operating system. This handler can be used for the case when: 1) an exception is raised in some task (process) for which no block _TRY/_UNLESS/_FINALLY/_END has been established, or 2) none of the handlers from the nested blocks provides treatment to the exception. This default exception handler is declared using the same exception mechanism (_TRY/ _UNLESS/_END block). When this wrapping block traps an exception, it translates it into a return code to make it compatible with the legacy applications (that expect error codes).

3.4 Supporting Real-Time Applications

Real-time applications demand predictable timing behavior. In embedded applications with timing constraints not only reliability and safe operation is demanded but also the ability to provide guarantees for timing behavior. As explained before, this implies that the worst-case execution time of the exception handler must be bounded.
Exception handlers are not predictable because their propagation rules allows them to propagate exceptions as much as required throughout an unpredictable large chain of exception handling blocks. In our design, we included dynamic propagation of exceptions, but using the mechanism in a restricted form, for applications with timing requirements. We establish by means of configuration, a bound in the depth of nesting (*nesting limits*) and in the number of active handlers, which indirectly provides a bound on the worst-case propagation time of the exceptions. This scheme can be statically analyzed with techniques such as those presented in [11].

3.4.1 Exceptions and Design-by-Contract Alignment

In the original design-by-contract semantics, a contract is fulfilled if all the preconditions, postconditions, and invariants are honored through the module execution. This original semantics specifies that if an exception occurs then the contract is considered broken. In this case, two semantics are offered, either repair the broken conditions and retry or shutdown the system gracefully.
Our design supports both semantics but in addition, it enables exceptions to exit module boundaries within the contract agreement. For instance, this could be honoring preconditions and invariants and so the caller module could retry calling this same module (or calling another) or even doing reparable (by the caller module) damage to preconditions. In summary, our construct enables a triple semantics for

exceptions in the context of design-by-contract: failure organized failure, retry, and reparable failures. These semantics are explained next.

Organized Failure: In the organized failure semantics the preconditions, and invariants are guaranteed upon the occurrence of the exception. However, the selected course of action is the orderly termination of the program, sending the exception outside the module.

Retry: Upon the evaluation of the conditions in an _UNLESS block (and potential corrective actions) it can be determined that both the invariants and the preconditions are still valid and a retry is possible. In such a case the _RETRY sentence can be used to retry the execution of the _TRY block.

Reparable Failures: In this case the occurrence of the exception could still honor preconditions and invariants or cause the preconditions to be invalid. However, provided that the calling code knows how to restore broken conditions the exception can be propagated to this code for a potential corrective action to keep the system running. In this case the alignment of modules and exception allows the common _FINALLY block to clean up any acquired resources (e.g. mutexes) to be released.

These two last semantics align exceptions with contract by keeping corrective actions within the limits of the contract (retry) or making the exception propagation as a valid exit option within the contract (reparable failures).

4 Comparative Study of Exception Handling Mechanisms

The aim of this section is to compare our exception handling mechanism against other well known mechanisms. This comparison is based on a previous comparative work from Garcia et-al [2]. In this work, Garcia et al identify common design features of exception handling mechanisms and compares different design solutions against them. These features are presented next.

Exception representation: Different choices exist to encode an exception. Exception representations can be classified as (a) symbols, (b) data objects or (c) full objects. Our mechanism supports symbols with parameters.

Checked exceptions: Two approaches are taken related to exception checking enforcement: checked exceptions and runtime exceptions as discussed in Section 2.3. Our mechanism does not support checked exceptions.

Clean Separation of concerns based on design by contract: Only Eiffel and our mechanism support a clear separation of concerns based on design by contract. Specifically, our mechanism aligns exceptions to module contracts enable recasting the interaction with the operating system as a contract.

Attachment of handlers: Exception handlers can be attached to different guarded regions, such as a). a statement, b). a block, c). a method, d). an object, e). a class, or f). an exception. In our mechanism exception handlers are attached blocks.

Propagation of exceptions: If a handler does not handle a specific exception, then an external handler is searched to handle the exception. This mechanism is known as *exception propagation*. In our design, the exception propagation is controlled by dynamically chaining guarded blocks. This chain is bounded in depth with a nesting

limit parameter. This parameter limits the number of active handlers (to limit the execution time of real-time applications).

Continuation of the control flow: After an exception is raised and the corresponding handler is executed the control cannot resume at the point of the exception. As explained before, our mechanism supports the termination and retry semantics.

Resource cleanup is a requirement of atomic transactions, which are important in many concurrent and real-time systems. For example, if an exception occurs within the critical section of a routine, it should release the critical section. Otherwise other processes wishing to use the critical section will be blocked indefinitely. We extended the exception with the _FINALLY code block that executes whether an exception occurs or not. This allows us to not keep a record of the locks retained by any section of code providing the programmer with a block where the locks can be released appropriately.

Concurrent exception handling: When concurrent exception handling is supported one or more exceptions can be raised concurrently. Our mechanism provides limited support for concurrent execution by automatic signaling to a supervisor task the termination of tasks by exception.

Table 1 shows the design aspect supported by Ada 95, C++, Java, Eiffel and our mechanism and Fig. 1 illustrates the complete semantics of our mechanism.

Taxonomy Aspects	Design Decisions	Ada 95	C++	Java	Eiffel	Ours
Exception Representation	Only Symbols	x				
	Symbols with parameter				x	X
	Objects		x	x		
Checked Exceptions	Unsupported	x			x	X
	Optional		x	x		
	Hybrid			x		
Clean Separation of concerns based on design by contract	Unsupported	x	x	x		
	Supported				x	X
Attachment of Handlers	Block	x	x	x		X
	Methods				x	
	Class	x			x	
Propagation of Exceptions	Automatic				x	
	Configurable					X
	Explicit	x	x	x		
Continuation of Control Flow	Termination	x	x	x	x	X
	Retry				x	X
Clean-up Actions	Explicit Propagations	x	x		x	
	Semi Automatic Clean-up			x	x	
	Specific Construct			x		X
Concurrent Exception Handling	Unsupported		x		x	
	Limited	x		x		X

Table 1. Comparison of Exception Handling Mechanism.

```
Int myCode(){
   _TRY { /* Protected Code Section */
    <  Program code (protected section).
       T1.- raise exception [RAISE(code,parameter)]
       T2.- verify retry identifier [RETRYCODE] >
    }
    _UNLESS { /* Exception Handling Code */
    <  Exception handling code:
       U1.- identify the exception code [EXCEPTION]
       U2.- obtain exception parameter [EXCEPARAM]
       U3.- retry protected code[RETRY(code)]
       U4.- abort operation [ABORT(code)]
       U5.- propagate the exception [default option] >
    }
    _FINALLY { /* Termination Code Section */
     <Termination Code>
    }
    _END
    return;/* return the protected block exit code */
   }
```

Fig 1. Proposed Scheme for Exception Handling.

5 Concluding Remarks

In this paper we presented a novel exception handling mechanism implemented in the kernel that supports multiple languages interfaces. This mechanism was developed along with a *C* language interface. We also discussed how our mechanism prevents the corruption of modular language structures (e.g. procedures and functions) when used along with exception handling blocks. This feature is aligned with the theory of design-by-contract enabling new semantics to support including exceptions as a recoverable exit path within contracts. We finally compared the multiple features of our mechanism with other well-known exception implementations.

References

[1] "How Airbags Work". http://auto.howstuffworks.com/airbag1.htm, as of 05/12/2005.

[2] A. F. García, C. M. F. Rubira, A. Romanovsky, J. Xu, "A comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software". *Journal of Systems and Software*. 59, pp. 197-222, 2001.

[3] R. Goldenberg,S. Saravanan,"*OpenVMS AXP Internals and Data Structures*",Digital Press, 1994.

[4] J. Gosling, B. Joy, y G. Steele. *The Java Language Specification*, Addison- Wesley, 1996.

[5] "Failure and Exceptions: A Conversation with James Gosling, Part II". Artima Developer, September 2003.

[6] "The Trouble with Checked Exceptions: A Conversation with Anders Hejlsberg, Part II". Artima Developer 18 de August 2003.

[7] G. Letwin, "*Inside OS/2*". Microsoft, Press, Redmond, Wa, 1988.

[8] B. Meyer, "*Object-Oriented Software Construction*", Prentice-Hall, 2nd Edition, 1997.

[9] B. Meyer, "*Eiffel: the language*", Prentice Hall Object-Oriented Series, 1992.

[10] D. A. Solomon "*Inside Windows NT Second Edition*". Microsoft Corporation, 1998.

[11] R. Chapman, A. Burns and A. Wellings, "Worst-Case Timing Analysis of Exception Handling in Ada". *Proc. Ada UK. Conference*, London 1993.