Abnormal Events Handling for Dependable Embedded Systems *

[a]Luis E. Leyva-del-Foyo, [b]Pedro Mejia-Alvarez, [c]Dionisio de Niz
[a]*Dpto. Computación, Universidad de Oriente, 90500, Santiago de Cuba, Cuba*
[b]*Sección de Computación, CINVESTAV-IPN, Av. I.P.N. 2508, México, D.F., 07300*
[c]*DESI, ITESO, Periférico Sur 8585, Tlaquepaque, Jal. México,*
E-mail: luisleyva@acm.org, pmejia@cs.cinvestav.mx, dionisio@iteso.mx

Abstract

In this paper, we analyze the difficulties of handling abnormal events. We introduce a framework that integrate the concepts of Design by Contract, exception safety and fault tolerance and from then, obtain a set of criterions for the design of a mechanism that integrate error code, exception and executable assertions for the handling of different types of abnormal events. From these criterions, a proposal for a novel exception mechanism adequate for C written embedded systems is presented. Finally, we analyze the advantages of our mechanism over the existing mechanisms and over other languages or previous extension to C.

1. Introduction

Many embedded systems must satisfy temporal and dependability requirements under high cost restrictions, limited storage and computational capacities. Examples of this type of systems exist in the automotive industry, medical equipments, industrial control systems and consumer devices.

The requirements of those embedded systems motivate the use of embedded operating systems (EOS) which do not use the traditional process model (such as the different UNIX variants or Windows NT). This process model is able to isolate the different activities (by using memory and input/output protection) so that the system reliability is to a certain extent determined by the trustworthiness of the operating system. Modern EOS provide basic services such as scheduling and a set of libraries to aid in the creation of applications but they do not support a process model, instead they provide a single physical address space for all the activities and the operating system does little to avoid that the programs interfere the one with the other or with the operating system. In this EOS, the programmer has total control over every aspect of the application and must provide all the routines to verify, to report and to treat all the abnormal conditions that can arise during the execution of an application.

Abnormal events handling is needed to ensure reliable and predictable operation of embedded systems in the presence of errors. Errors can result from hardware failures, software bugs, discrepancies between the external environment and the internal representation, or from improper timing and synchronization.

Traditionally the mechanism of return of error codes has been used for the handling of abnormal events. However, the well-known drawbacks of this method [7] brought as consequence the introduction of exceptions handling mechanisms with the promise to separate the error handling code from the normal code [13]. The introduction of exception constructs has helped programmers to improve the reliability of software, however the semantics (and the supporting syntax) of current exception constructs is still too loose leaving plenty of room for their misuse. For instance, a common misuse is to use exceptions to report back rather regular events. Yet another common misuse is to catch exceptions with an empty catch block (ignoring the exception). This misuse is a symptom of improper support for the separation of concerns in current exception handling mechanisms. Perhaps the worse consequence of the lack of a proper separation of concerns is the negative correlation between correctness and reliability, commonly observed in software systems as identified by Parnas in [25]. This correlation suggests that when a software system needs to observe both correctness and reliability, increasing the emphasis in one decreases the quality of the other.

The previous exception constructs results in an excessive increase of the code associated to the handling of abnormal events. This also increases the complexity and costs of the system without contributing significantly to the correction and reliability requirements of the application. In this context, we believe that it is important the existence of a general conceptual framework for the handling of all errors types. This framework should promote the use of the adequate mechanisms and a correct separation of responsibilities among the different software elements and among different sections of codes. Also, appropriate language constructs are necessary to allow the direct implementation and enforcement of this framework. In this context, the main contributions of our work are:

* This research work has been supported in part by NSF-CONACyT project 42449-Y, Mexico.

- A conceptual framework that integrates design by contract, exception safety and fault tolerance. This framework provides a clear separation of responsibilities for abnormal handling and a right balance between the correctness and reliability requirements.
- The proposal of a set of criterions for the design and use of an abnormal events handling mechanism based on this conceptual framework.
- The design of a novel exception handling mechanism based on these criterions with a supporting syntax and semantic that prevents important misuses and adapted for C written embedded systems.

This work is organized as follows. In section 2, the general difficulties associated to the handling of abnormal events are presented. Section 3 introduces our conceptual framework for the handling of abnormal events and its derived criterions for the design of an abnormal event handling mechanism. Section 4 proposes an exception handling mechanism based on these criterions for C based embedded systems and section 5 highlights the differences of the mechanism proposed with some other existing mechanisms. Finally, section 6 presents some concluding remarks.

2. Abnormal event handling difficulties

2.1. Hierarchical Architecture Drawbacks

Usually software is structured in layers to provide an easier management of its complexity. Each layer is constructed using the information hiding principle [24] and communicates only with adjacent layers through well defined interfaces. Each layer implements a level of abstraction or a virtual machine and these levels form a hierarchy at which the highest level implements an specific functionality of the application. As one lowers in levels, the abstraction diminishes until arriving at the lowest level where generally there is a direct interaction with the hardware. Nevertheless, this structure makes difficult those aspects requiring cooperation among several levels, such as the handling of abnormal events.

The problem with this hierarchical structure is that, since most of these abnormal events are detected at the inferior levels, it is not possible to carry out appropriate recovery procedures. This is because at these inferior levels there is neither sufficient global information nor specific context of the application. The current existing solution is to allow the interfaces between the modules to communicate the errors occurred among them [25]. This are done in a way that these errors propagate from the lower levels, where those were detected, up to the higher levels at which appropriate decisions can be taken to cope with error treatment [13].

2.2. Returning status or error value

The traditional mechanism of C to carry out abnormal events handling consists of returning an status or error value on each function. This code must be verified explicitly at each level in the hierarchy to determine if the normal processing should continue, if it is necessary to give treatment or to return immediately to the invoking procedure propagating the error.

The problem with this approach is that, when an error is detected on functions located at the middle of the hierarchy, it rarely can be correctly treated, so it must be propagated. These errors are usually generated at lower level functions and they are not directly related to the logic of the current computation, thus they cannot be treated by the function. These errors are named *tramp errors* [2] [26]. The presence of tramp errors in the interface of a function breaks with the function's abstraction, because it reveals information referring to its implementation. Even worse is the fact that the invoking code, cannot avoid those implementation details, which are irrelevant to its logic of operation. This is because the invoking code has the responsibility to propagate the error explicitly to the higher levels. The presence of this propagation code, mixed with the function's code, introduces a tight data coupling among layers which complicates reusability. Additionally, programmers tend to ignore this extra code because the propagation code is not within the normal logic of the program and it makes it difficult [7][21].

2.3. Using Exception Handling Mechanisms

With the purpose of solving the above mentioned problems, several specific control structures, named exception handling mechanisms [13], were developed to handle abnormal events. Modern programming languages, such as Ada[16], Eiffel[19], C++[31], Java[11], and C#[14] provide support for this constructs.

It is argued that exceptions are superior to the return of error codes because (1) they provide a clear separation of the error handling code from the normal code [10][32], (2) do not require a frequent verification and propagation of the error values [10] and (3) the program cannot ignore the errors returned by the lower level functions [21][27][32]. All those advantages should lead to the construction of software with lower complexity, easier to read, more efficient and hence with less bugs [13].

However, the experience has demonstrated that the excessive use of exceptions leads to the development of programs plagued with try-catch blocks which in most cases: (1) handle rather normal situations [32], (2) convert

[2] In analogy with the term "*tramp data*" in the structured analysis: a piece of information that shuffles aimlessly around a system, unwanted by — and meaningless to — most of the modules through which it passes [23].

exceptions among abstraction levels while they are being propagated, or (3) even catch the exceptions ignoring its treatment (to avoid compiler errors). These **try-catch** sentences are more difficult to read and much less efficient at run time that the alternative **if-else**. In addition, it is easier to introduce bugs due to the presence of implicit control transfers.

All current research efforts dedicated to characterize the situations for which exceptions must be used instead of the returning of error codes, agree in that they must be used only under exceptional or rare conditions. Nowadays the most elaborated definitions of exceptional conditions are based on the design by contract theory [19]. Under this theory an exceptional condition is that which does not fulfil the contract, or that on which it is not possible to fulfil the contract [32]. Although this definitions are precise, they are stated in terms of a formal model of the software system and therefore they do not give an engineering criteria to decide what must be (and what must not be) included into the contract. So, in practice they are not always very useful.

2.4. Difficulties associated to embedded system

2.4.1. Support for C-written Programs

Nowadays the C language continues being the language of choice for a broad range of embedded systems. The great success of C is given by its combination of low level features, which allows a high degree of direct hardware control needed for embedded systems, along with its processor independence. This characteristics contrast with the higher level and more secure features of others languages, which overcome its ability to easily control the hardware. On the other hand, C++, the object oriented superset of C, which provides native support for exception handling, is increasingly popular for embedded systems. However, some of its new features reduce the efficiency of executable programs.

The lack of a structured exception handling mechanism in C is an important drawback for the creation of reliable systems. So, we argue that the incorporation of a well-designed exception mechanism to the C language allows an adequate balanced between efficiency, portability and reliability for the development of reliable embedded systems.

2.4.2. Support for fault tolerance

Given the dependability requirements of the embedded systems, the exceptions mechanism must provide adequate support for fault tolerance [8]. This aspect introduces particular characteristics for embedded systems which are exposed next.

Minimum redundancy and diversity of software. The provision of fault tolerance does not come without cost. This cost depends mainly on the type of redundancy and diversity used. Due to the cost restrictions, in some cases hardware redundancy is not an option and then the software must take care of the fault tolerance. Nevertheless, due to the increase in the amount of code and the overhead in the execution times, it will be necessary more powerful processors and more memory with the corresponding increase in the hardware cost. The challenge is then to provide suitable levels of fault tolerance with minimum cost increases.

Error recovery. This takes care of bringing the system from an erroneous state, back to a consistent state. Error recovery may be carried out by correcting the damaged state (known as *forward error recovery*) or by returning the system to a previous known consistent state (known as *backward error recovery*) [4]. The backward error recovery has the advantage of providing a transparent implementation which provides an efficient recovery from unpredictable errors. For this reasons, backward error recovery is used in most general purpose systems. Nevertheless, its main drawback is the requirement of saving the system state in a set of *recovery points*. This extra state may be a burden in resource-constrained system such as embedded systems. In addition, the state of embedded systems may also include the actuation to the external environment that cannot be undone without consequences. As a result, an abnormal event handling mechanism for embedded systems must provide support for backward and forward error recovery.

Damage confinement. After an error is detected, it must be verified to what extent the system has been corrupted by the error. In embedded systems, after detecting an error, it cannot be guaranteed that the caused damages have been restricted to certain areas of the system. This is because of, the absence of security features in the C language, the use of EOS, the use of hardware without memory protection and the use of direct programming over the bare hardware.

2.4.3. Compatibility and legacy systems support

The mechanism for handling abnormal situations must allow: (1) A maximum portability, allowing the implementation of any construction of the language as a C extension supported by the compiler (to maximize efficiency), or as pre-processor macros (for it use with existing compilers), and (2) A maximum compatibility, allowing the reuse of legacy C code that has not been written specifically for this mechanism.

3. Criteria for abnormal event handling

In this section we first discuss design by contract, the formal framework under which we provide a clean separation of concerns. Then we provide a classification of the types of abnormal events and its dynamics, which allows us to identify the requirements and mechanisms necessary to handle each of these events and the responsibilities of the different software elements for handling those events. After that, we analyse exceptions safety and fault tolerance to identify the issues that allows

us an adequate achievement of dependability and correctness requirements. From this analysis, we provide design criterions for the treatment of abnormal events for embedded systems.

3.1. Separations of Responsibilities

At the core of separation of concerns is modularization. This implies that a good separation of concerns cannot be achieved without an strong emphasis in a clear modularization. To achieve this we base our modularization semantics and syntax in the Design by Contract Theory [19].

Design by contract supports a formal separation of responsibilities among software elements (objects, components, function or code sections). According to this theory, a software system is a set of software elements that interact on the base of a well defined specification of mutual obligations which constitute a *contract* between the *supplier element* (the one whose methods or routines are invoked) and the *client element* (the one that invokes the methods or routines). This contract is composed of a) the *preconditions* guaranteed by the client element, before a routine of the supplier is executed; b) the *postconditions* guaranteed by the supplier when the routine ends; and c) the *invariants* or conditions that applies to the entire element that characterize its consistency and integrity properties. If the client element assures the satisfaction of the preconditions, then the supplier element assures the satisfaction of the postconditions and invariants.

Using the notation defined in [15] the responsibilities assigned to a *Code* section (in terms of the contract) can be specified as:

$\{precondition \wedge invariant\}\ \ Code\ \ \{postcondition \wedge invariant\}$

The separation of responsibilities between the *client* and *supplier* code sections in a program is illustrated in Table 1. The *relyer* is the software element that assumes that a condition (precondition, postcondition or invariant) is true, while the *ensurer* is the element that has the responsibility of guarantee the condition.

Two code sections are related by a condition if one is *ensurer* and the other is *relyer* of this condition. An ensurer code has *fulfilled* the condition, if it transfers the control to a relyer code, when the condition that relates them is true. If an ensurer code evaluates the condition that relates it with a relyer, in order to make sure that it is satisfied, it is said that the client has *tested* the condition. On the contrary, if the relyer code evaluates the condition that relates it with an ensurer, with the purpose of checking its validity, it is said that it has *checked* the condition[3].

Table 1. Conditions related relationship between code sections

	Precondition	**Postcondition**	**Invariant**
Relyer	Supplier	Client	Supplier
Ensurer	Client	Supplier	Supplier

3.2. Error Semantics & Constructs Alignment

We define *abnormal events* as those that may arise during the execution of an operation and that are related with this operation[4], but require an immediate change of the normal course of execution for its treatment. The omission of this treatment prevents the fulfilling of the dependability requirements, therefore, any classification of these abnormal events should be made in the context of the *dependability impairments* [17]. Here we can distinguish the following concepts: (a) *failure*, defined as the deviation of the service delivered by the system from the behavior specified in their requirements, (b) *error*, defined as the system state that is liable to lead to a failure, and (c) *fault,* defined as the identified or hypothetical cause of an error. It is known that there are faults after errors are detected and the errors can be spread, to produce other errors [17]. In this context, we classify the following types of abnormal events and provide different strategies and priorities for their handling:

Software error: it is an invalid state at which the system can enter as a result of a *software fault* or *bug*. These faults are persistent and occur because of design or implementation errors. Although they should never exist, it is impossible to avoid them. In consequence, a dependable system should have some strategy to deal with them. According to the form at which the bugs are manifested it is common to distinguish two types [10]: *heisenbugs* that lead to a transient, intermittent software errors; and *bohrbugs* that are manifested in a reliable way under a set of well defined conditions. When we deal with software faults the priority is to detect (and to correct) them as soon as possible (before the system is deployed).

Application error: it is an invalid state of a system caused by circumstances that arise in a justifiable and unavoidable way during its execution. This error can be the failure of another external component or the fail of the required service. We distinguish two types of application errors:

- *Incident*: It is a situation that arise during the execution of an operation (inside the logic of the application) and which it is completely foreseeable. Only few incidents are possible and they should be completely enumerated.
- *Emergency*: It is an uncommon and not very frequent situation that, although it could be anticipated, does not match with the current abstraction level (tramp error). It is not possible to foresee all potential emergencies

[3]The concepts of *test*, *check* and *fulfill* are generalizations for conditions of the concepts defined in [20] for preconditions.

[4]This requirement discards the asynchronous events (not related) like the interrupts or the UNIX signals.

that could arise during the execution of a software element.

The incidents must be treated directly by the calling code, therefore the priority when facing them is to provide efficient mechanisms for their treatment. In the other hand, when an emergency arises, it is not possible to provide means to solve the problem within reach of the immediate caller. Therefore, the priority is to allow the safe propagation through all the software elements located among the point at which it is detected and the point at which it can be treated; as well as, to allow the safe recovery of all these intermediate elements in the face of any abrupt interruption due to this propagation.

Criterion 1: Support for multiple error-handling mechanisms. *There is not a unique general mechanism for the handling of all the types of abnormal events.*

Depending on the error type different mechanisms must be used:

Incidents handling: The most effective form to report them is using a return of error codes. This is because the calling code can perform its treatment efficiently using the normal language instructions for conditional control transfers.

Emergency handling: The most effective form to report and propagate them is by the use of an *exception mechanism*. This is because the exception mechanism is specially designed to cut through the calling stack to reach the appropriate element where the error can be treated. The exception mechanism is restricted only to the task of propagating abnormal events. This is the only aspect where it is less expensive than the other alternatives.

Software error handling: In this case, the most effective way is the use of *executable assertions* [30]. The non-fulfillment of these should invoke a global handler to record enough information to fix the bug and perform a fail-safe or a reset. The use of executable assertions to handle software errors simplifies greatly the design of the exception mechanism and of the error treatment code, hence reducing code size. Without damage confinement features (subsection 2.4.2), the use of executable assertions diminishes the possibility of spreading the error and increases the possibility of system recovery (because the same recovery code or its environment could have been corrupted). Also, it promotes the construction of bug-free software. Table 2 summarizes the types of abnormal events and its correspondence with the priorities and the language mechanism used for its handling.

By providing a correct identification and classification of possible abnormal events that can arise in the different system components, it is possible to identify precisely the software elements responsible for handling such events and the mechanism required for its reporting. This avoids an excessive growth of the code needed for errors handling and allows the construction of dependable systems with minimum costs increments.

Table 2. Integrated Strategy for Abnormal Events handling

Type of Abnormal Situations		**Priority to deal with it**	**Language Mechanisms**
Software Error		Early detection / fail-safe or reset	Executable assertions
Appli-cation Error	Emergency	Safe propagation and recovery	Structured exceptions
	Incidence	error treatment	Error codes/ Normal construct

Criterion 2: Single point of correctness or minimum code redundancy*: avoid having two code sections that rely in, and ensure the same conditions.*

This principle is a consequence of the application of the design by contract (jointly with the verification of the conditions), to allow an early detection of software faults. This is because, in this way the introduction of heisenbugs is minimized and more bugs may be manifested as bohrbugs. This principle discards the use of the N-version programming approach to fault tolerance [3] which is consistent with the resources restrictions for embedded systems.

3.3. Abnormal events characterization

The identification and classification of abnormal events must be performed at early design stages for all the system components. This allow us to correctly define the responsibilities for each system component in their handling of abnormal events.

The concepts introduced in section 3.1 help us to formally define a *software error* as the non-fulfillment of a condition, that is detected by a checking. This allows a clear separation of responsibilities for detecting and reporting software errors from those of detecting and reporting application errors. The first is responsibility of the *relyer* code, while the second is responsibility of the *ensurer*. Although this separation of responsibilities (given in terms of the contract) is important, it does not specify the conditions of the contract. The specification of the conditions of the contract is obtained using engineering criteria that take into account the context of the application and its abstraction level.

Application dependency. The classification of a concrete event depends on the application. For example, in a dynamic system that operates in an environment where its resources are constantly changing, the impossibility of assigning a resource (such as the memory) is considered an emergency. However, in applications executing in predictable environments, where the resources (including memory) need to be sized at design-time, the impossibility of assigning resources constitutes a software error.

Abstraction level dependency. The classification of a concrete event is local to a software element. For example, at the I/O level, a failure when reading some external storage device is an incident (it is perfectly foregone and inside the logic of the operation). The immediate superior level (I/O logic) must be prepared to deal with this situation, by retrying the operation (hoping that the cause has been some transitory failure). However, if this situation persists, it has to be reported as an emergency so that it is propagated up to the point where enough contextual information exists. If this higher level of abstraction was trying to locate information for the first time, it is possible that the error was due to that the storage media had not been already introduced, or that it was due to an invalid media. Again, this can be considered a situation inside the application logic and therefore should be reported as an incident. The immediate upper level is prepared to manage it, for example, by requesting the insertion of the media.

Engineering Criteria: As more general purpose is a software element, it is more reasonable to state that the different abnormal events are expected and are inside the caller logic. In other words, at the lowest levels of the architecture the abnormal events are incidents and therefore they should be reported by returning error codes. The upper levels tend to be application oriented and it is reasonable to state that the incidents (error codes) reported by the lowest levels now are transformed into emergencies (exceptions), so that they travel in an implicit way to the higher levels, avoiding tramp errors. Lastly, when these errors are being reported at the highest level of the application (where one must treat them) they should again turn into incidents (error return codes) according to this level of abstraction.

Criterion 3: conversion among errors types: *The mechanism used for abnormal event handling should support the conversion (preferably automatically) among different errors types (exceptions, return codes and assertions violation).*

This feature would eliminate the great amount of the code needed to translate one error type to another one (change of abstraction level). This code is responsible for many of the **try-catch** construction in a program. This makes easier the understanding, the maintenance and the reusability of the code.

3.4. Understanding Exception Correctness

It is well know the fact that writing correct code in presence of exceptions is difficult [5][27]. The main problem is that exceptions hide the control transfers that break the explicit control flow of the operations. When an exception is raised on a deeply nested service, all functions in the invocation chain are abruptly interrupted. This interruption may leave the data structures associated to these functions in an inconsistent state or it may cause resource leaks due to the skipping of the code where such resources were released (e.g. memory).

To specify the software behavior in presence of exceptions we will adopt the Abrahams guarantees [1]. These guarantees are:

- *Basic guarantee*: No resources are leaked; software elements remain in a valid although not predictable state.
- *Strong guarantee*: The state of the program remains without changes. This guarantee always implies a global commit-or-rollback semantics.
- *Non-Fail*: The operation never raises an exception.

Here it is important to emphasize that (1) the impossibility to offer at least the basic guarantee is considered a software error; (2) although not always is possible to provide the non-fail guarantee, in some operations it is a mandatory requirement (e.g., deallocation and swap function). Without this requirement other operations cannot even provide the basic guarantee [33]; (3) the strong guarantee is different than the other guarantees because it is the only one which is dependent on the application requirements.

Criterion 4: Separation of concerns for exception correctness*: The exception mechanism should provide explicit support for the attainment of the basic guarantee.*

Aligning this support with the Design by Contract involve two independent aspects:

a) The recovery of the local invariants (consistent state).
b) The preservation of the system global invariants (i.e., absence of resources leaks, absence of deadlock).

The explicit distinction of the code sections responsible for each one of these aspects is important because:

1. Local invariant recovery code (a) has to be executed only when a code section is aborted abruptly by an exception, while the global invariant recovery (b) must be executed independently of the way that the code section ends.
2. It allows for restricting the execution order so that (a) is executed first and (b) after that, so that the resources are assigned when (a) is carried out.
3. The logic of preservation of the local invariants is specific of the software element and cannot be generalized. On the contrary, the logic of preservation of the global invariants is more general and can be feasibly automated (e.g., garbage collection, monitor locks release).
4. It allows us to establish the local invariants as precondition of the code that preserve the global invariants enabling its verification at runtime.

The last aspect is of paramount important for schemes that automate the preservation of global invariants. For example, in Java the exceptions release all the locks from the invoked object when a synchronized method raises an exception to the invoker [11] while local invariants are

not enforced. Consequently, Java programs are prone to leave objects in an inconsistent state [9].

Criterion 5: Guarantee for exception correctness. *The codes in charge of local and global invariants must offer the non-fail guarantee. If this condition is not met, then we are in presence of a software error*.

Without this criterion it is impossible to offer the basic exception guarantee. For example, if an exception in Java is thrown inside a **catch** or a **finally** block, it is propagated to the outer **catch** block (aborting the recovery of the local or global invariants). This fact and the impossibility to guarantee non-fail operations in Java, preclude writing exception safe code [33].

3.5. Exception Handling Decomposition

The exception mechanism has to provide adequate support for a system fault tolerance scheme (subsección 2.4.2). In *error handling* we differentiate *error recover* from *error treatment*. Error recovery returns the system to a consistent state. Error treatment is in charge of satisfying, as best as possible, the service requested (maybe with an alternative algorithm or by allowing an acceptable degradation), so that the system can continue with its operation. When treatment is provided, many times it is necessary to recognize the cause of the error and if possible to carry out the necessary actions to avoid that it happens again.

The distinction between recovery and treatment is important for the following reasons:

1. It allows for establishing a correspondence between the fault tolerance and the exceptions guarantees: the error recovery is responsible for assuring the basic guarantee, while the error treatment is responsible for assuring any additional guarantee.
2. The recovery logic is straightforward and independent of the semantics of the application's upper abstraction levels. Precisely, it is local to the element. Often also it will be independent of the error (or exception).
3. The treatment logic is specific and it depends on the error and its cause, as well as on the context and requirements of the application. It is conditioned by design decisions that can facilitate, obstruct or even make it impossible (by lack of redundancy or diversity). Therefore, this logic is inside (and it is integral and inseparable part) of the application logic.

This allows us to clearly separate the responsibilities of the code associated to the constructions of the exception mechanism, from those of the code associated to the normal (conditional, iterative) control structures of the language.

Criterion 6: Separation of concerns for error handling. *Error recovery must be set apart from error treatment. The structures of the exception mechanism should be responsible only of the recovery logic, transferring the control to the normal code for their treatment*.

This separation of concerns is important for the following reasons:

1. It allows the application of the criteria of single point of correctness (sub-section 3.1) inside a software element, for extracting the recovery code for all the exceptions into a single place.
2. It shows that the separation between the normal application logic (free of errors) and the error handling logic [10] is an erroneous separation of concerns. The correct separation of concerns is between the code of error recovery and the code of error treatment.

4. Exception Handling Mechanism

In this section we introduce the design of an exception handling mechanism compliant with the criteria presented in Section 3. We first present the syntax and semantics of the language constructs and then discuss how it supports different aspects of the framework.

4.1. Exception Handling Construct

Our exception handling construct consist of three code blocks that we identify as: **_TRY**, **_UNLESS**, and **_FINALLY** blocks (see Figure 1). A **_TRY** block encapsulates the code inside of which the exception could occur. The exception is signaled with the **_RAISE**() function passing a failure code (known as exception) and a failure parameter, transferring the control to the **_UNLESS** block. The **_UNLESS** block encapsulates the code that takes care of the exception. Lastly, the **_FINALLY** block is executed after the **_TRY** block (when no exception was raised) or after the **_UNLESS** block (when exception was raised). The **_FINALLY** and **_UNLESS** block are optional. This exception construct may be nested.

Inside an **_UNLESS** block, the code can query the **_EXCEPTION** variable to decide what to do for the different exceptions that can occur. If the **_UNLESS** block code decides not to do anything to treat the exception, it is propagated implicitly to the outer protected block. To prevent this implicit propagation, the code in the **_UNLESS** block has three options: retry the protected block (**_RETRY**), abort the propagation with a status indication (**_ABORT**), or translate the exceptions code into another one and keep propagating it (**_XTRANS**).

```
Int myCode()
{
 _TRY { /* Protected Code Block */
    Regular code where the following may occur:
     - raise an exception
       [_RAISE(code,parameter)]
```

```
      - abandon protected block [_LEAVE(code)]
      - verify retry identifier [_RETRYCODE]
  }
  _UNLESS { /* Error Recovery Code Block */
    Error recovery, can do the following:
     - identify the exception code [_EXCEPTION]
     - obtain exception parameter [_EXCEPARAM]
     - reiterate protected code[_RETRY(code)]
     - abort operation [_ABORT(code)]
     - Translate exception [_XTRANS(code)]
     - propagate the exception [default option]
  }
  _FINALLY { /* Termination Code Block */
    It is executed with or without exception.
  }
  _END
  return;/* return protected block exit code */
}
```

Figure 1. Exception Handling Mechanism

4.2. Semantics of the exception mechanism

For an exception construction *x*, let us define Try_x, $Unless_x$ and $Finally_x$ as the code blocks associated to **_TRY**, **_UNLESS** and **_FINALLY** in that order, and $Xall_x$ as the code associated to the complete construction. Let PRE_x and $POST_x$ be the corresponding precondition and postcondition associated to *x*. Let *e* be the software element on witch *x* operate and let INV_e be the invariant that must guarantee all the code that modifies *e*.

If (according to subsection 3.4) the local invariant *LINVe*, that defines the consistency of *e*, is differentiated from the system global invariant *GINV*, that is responsibility of all the code, then:

$$INV_e = LINV_e \wedge GINV$$

Moreover, if (according to section 3.1) the *x* post-condition, in case of success $SPOST_x$ is differentiated from the *x* postcondition for unsuccess, returning an error status code $UPOST_x$, then:

$$POST_x = SPOST_x \vee UPOST_x$$

Based on the above definitions, the specification of the code associated to the whole construction $Xall_x$ can be stated in terms of the presupposed initial state and of the final state that has to be guaranteed, as follows:

In case of successful exit:

$$\{ PRE_x \wedge INV_e \} \; Xall_x \; \{ SPOST_x \wedge INV_e \}$$

In case of exit with an error status code (incident):

$$\{ PRE_x \wedge INV_e \} \; Xall_x \; \{ UPOST_x \wedge INV_e \}$$

In case of raise an exception (emergency):

$$\{ PRE_x \wedge \mathrm{INV_e} \} \; Xall_x \; \{ INV_e \}$$

In the achievement of each one of the previous final states, the responsibilities of each one of the blocks of the construction x, is defined as follows:

For Try_x:

$$\{ PRE_x \wedge LINV_e \wedge GINV \} \; Try_x \; \{ SPOST_x \wedge LINV_e \}$$

For $Unless_x$:

In case of implicit or by **_XTRANS**() propagation:

$$\{ True \} \; Unless_x \; \{ LINV_e \}$$

In case it prepares for treatment by means of **_RETRY**():

$$\{ True \} \; Unless_x \; \{ LINV_e \wedge PRE_x \}$$

In case it prepares for treatment by means of **_ABORT**():

$$\{ True \} \; Unless_x \; \{ LINV_e \wedge UPOST_x \}$$

For $Finally_x$:

$$\{ LINV_x \} \; Finally \; \{ LINV_e \wedge GING \}$$

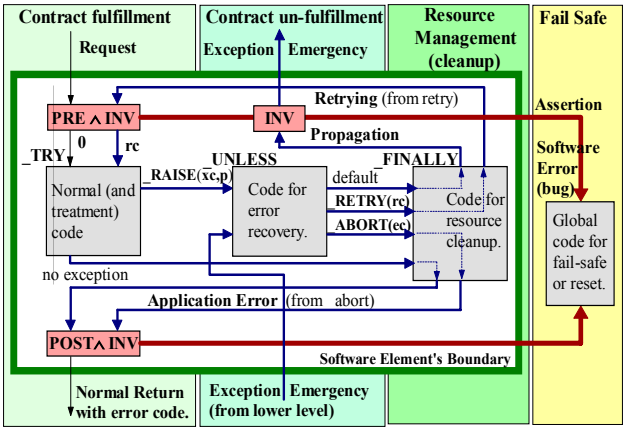

Figure 2. Separations of Responsibilities of the Mechanism

4.3. Separations of Responsibilities

Figure 2 illustrates the mechanism semantics and emphasizes the separations of responsibilities among different code blocks, and among the mechanisms (assertions, exceptions and normal language constructions). The thin lines depict the control flow of the normal language instructions. The thick lines represent the transferences associated to the fail of executable assertions (software error detection). These assertions explicitly insert the contracts of section 4.2 in the corresponding block and check them automatically at runtime. They provide support for the customization of the application response to *software errors* by setting an applications specific global handler (right-most part of Figure 2). In an embedded application, this handler takes the system to a fail-safe mode and then triggers a system reset. In the development phase, this handler is an ideal place to hard code a permanent debugger break point. The rest of the lines represent the control transfer associated to **_TRY**, **_UNLESS** and **_FINALLY** code blocks of the structured exception mechanism. **Such control transfers are caused by _RAISE**(), _RETRY() or **_ABORT**().

Table 3 summarizes the separation of concerns associated to the different blocks of the exception constructs from the perspective of the fault tolerance, exception safety and design by contract.

Table 3. Support for the separation of concerns for Exception correctness and Error handling in the exception mechanism

Fault	Exception	Design By	Implementation

Tolerance	Correctness	Contract	Mechanism
Error Recovery	Basic Guarantee	$LINV_e$	**_UNLESS**
		GINV	**_FINALLY**
Error Treatment	Above Basic Guarantee	$POST_x$	**_RETRY/ ABORT** Application Logic

4.4. Abnormal events characterization

Since errors need to have different semantics at different levels of abstraction, our mechanism provides functions that convert exceptions to other exception codes or to return codes. To translate an exception code into another one, to be sent to a higher-level module, we use the function **_XTRANS**(). The **_ABORT**() command in the **_UNLESS** block, allows the translation of the exception (or emergency) into an error exit code (or incidents), at the appropriate abstraction level. This exit code may be obtained after the **_END** sentences using the **_EXITCODE** command. Another important feature of our mechanism is the use of **_TRYERROR**, instead of **_TRY**, to perform an automatic conversion of all exceptions into error exit codes. **_TRYERROR** avoids the use of many **_UNLESS** blocks, which have the conversion as its unique purpose, thus making the code clearer. This option also allows the encapsulation of software elements that raise exceptions and must be used by legacy C code (subsection 2.4.3).

4.5. Exception handling decomposition

In Figure 3, a rearranged and simplified drawing is presented to show how the consistent use of this exception mechanism and the criterions of section **3** allow the creation of a *fault-tolerant capable element* (FTCE). The FTCE is an adaptation of the *ideal fault-tolerant component* of Anderson and Lee [4]. The element accepts service requests and, if necessary, calls the services of other elements before producing a response. It can signal two types of faults: emergency (reported by exceptions) and software error (reported by assertions). However, it can not support a *full fault tolerance* nor a *graceful degradation* approach that tries to keep running after a software error has been detected. To provide adequate support for embedded systems, this type of error can be handled only through the *fail safe* approach (subsection 2.4.2).

Our FTCE enforce a clear separation of concerns between code sections: the responsibility of the normal code is to ensure the routine contract but not perform the recovery from the exception. The exception code is not intended to guarantee the contract; instead its purpose is only to perform the error recovery. In other words, it must restore the invariants and execute **_ABORT** (for forward error recovery) or restore also the preconditions and execute **_RETRY** (for backward error recovery). In any case, the control flow goes to the "normal" processing code to treat the error so that it performs a new intent to fulfill the contract. In otherwise, in case of failure (without retrying or aborting) the exception is raised at the caller.

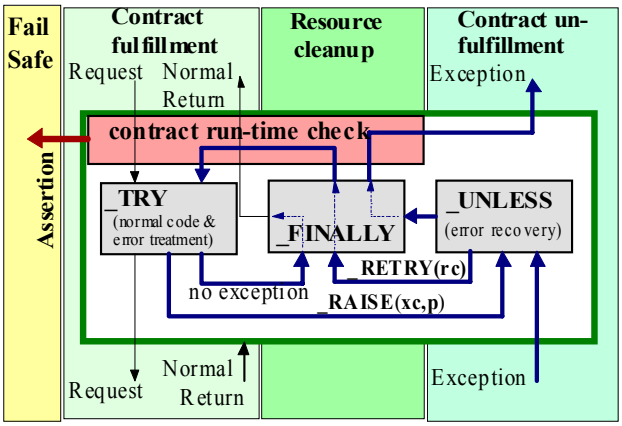

Figure 3. Fault-Tolerant Capable Element (FTCE).

4.6. Support for exception correctness

The support for exception correctness is achieved by providing explicit support for:

- Criterion 4: As specified in sections 4.2, the recovery of the local invariant is the only responsibility of the **_UNLESS** block, while the **_FINALLY** block is in charge of the global invariant and has as precondition the local invariant.
- Criterion 5: The **_RAISE**() sentences can be invoked only inside a **_TRY** block. The **_XTRANS**() sentence is provided to allow the **_UNLESS** block to translate an exception code while propagating it. This sentence only can be used inside an **_UNLESS** block. A **_FINALLY** block cannot raise or translate an exception.

The executable assertion mechanism captures as a software error (subsectión 3.2) any intent of propagating a new exception from an **_UNLESS** or **_FINALLY** block, as well as any intent of leaving an **_UNLESS** block without restoring the local invariant.

5. Contrast with related works

This section emphasizes the differences of the exceptions mechanism proposed, with those of other object oriented languages of the C family or based on the Design by Contract as well as with other extensions to the C language.

5.1. Contrast with other language mechanisms

5.1.1. Object Oriented C Family Languages

The differences between the mechanism proposed and the mechanisms present in the objects oriented languages derived from C (Java, C++ or C #) are the following:

- By default, the **catch** clause (of C++, Java or C#) considers that its code block provides treatment to the error and therefore does not continue its propagation. In contrast, in our mechanism the **_UNLESS** block should never treat the error. If it does not end explicitly, the exception is propagated to the external block.
- The **catch** construction does not discern between the code for error recovery and the code for error treatment. In contrast, the **_UNLESS** block only has the responsibility for error treatment and not for error recovery.
- Multiple **catch** clauses may exist and all of them are qualified. When an exception is thrown only the one that first match is executed. In contrast, there is only one **_UNLESS** for each **_TRY** that is executed for any exception.
- The blocks associated to the **catch** (Java, C++, C#) and to the **finally** (Java or C#) are not forced to offer the non-fail guarantee. In contrast, the blocks associated to the **_UNLESS** and to the **_FINALLY** must offer the non-fail guarantee. Any intent of propagating an exception outside of them, is considered a software fault captured by the executable assertions[5].

5.1.2. The Eiffel Language

Eiffel exceptions are based on the principle of the design by contract. However, our mechanism follows a different approach.

- In Eiffel the exceptions indicate software errors (a condition violation in a test). In our scheme these errors are not indicated by exceptions, but by invoking an executable assertions handler.
- The Eiffel exceptions are raised in an implicit way by the run-time support system. In our case, this is an explicit responsibility of the ensurer code using **_RAISE** ().
- The code for **rescue** in Eiffel can only make explicit transfer to the beginning of the protected code (restoring the invariants and the precondition) therefore it does not provide appropriate support for forward error recovery. Alternatively, our mechanism provides explicit support for forward and backward error recovery.
- The **rescue** clause only offers explicit support for the preservation of the local invariants in presence of exceptions. In contrast, our mechanism offers explicit support for preserving the local and the global invariants.

[5] For the **_FINALLY** case, this is equivalent a **terminate**() call in C++ if during an exception stack unwind, an exception is propagated from a destructor. The use of *resource acquisition is initialization* [31] place the destructor in the same role in C++ exception mechanism than that of **_FINALLY**. Both are variants of the *responsibility management pattern under exceptions* [22].

5.2. Contrast with other extension to C

Although several C extensions for exceptions handling have been introduced in the literature, with few exceptions [10], all of them have been designed specifically for desktop systems. The work in [18] demonstrated that exceptions can be added to C without language changes (using only standard preprocessor features). Since then, many other introduced exceptions using the same approach [18], or using minor language extension [10]. Many of them have semantics similar to: Ada [10], Eiffel [6] and Java/C++ [29][34]. Some also include support for the resume model of exception handling and for asynchronous signal handling [2][10][6]. The work en [26] presents a higher level transaction approach to error handling, however it is not appropriate for application dependent recovery. The work in [21] is the only one based on an a error handling classification scheme. It defines fault (our software fault) and failures (our emergency), however its framework does not support incidents, does not integrate fault tolerance and exception safety, and does not define the precise responsibilities of the exceptions blocks. Its resulting exception mechanism is less disciplined and may produce the same problems of the traditional method, due to the excessive use of exceptions (see subsection 2.3).

6. Conclusions

In this paper, we analyzed the difficulties of handling abnormal events and provided a classification for the different types of those events. From this classification we proposed an strategy to integrate error codes, exceptions and executable assertions for the handling of different types of abnormal events, along with a support for the conversion of error types.

We introduced a framework that integrates the concepts of Design by Contract, exception safety and fault tolerance. From this framework, we obtained a set of engineering criterions for the design of a mechanism of abnormal event handling. This engineering criterions allows us to determine the conditions of the contract and the classification and transformation of all types of errors in the context of the application and the abstraction level.

This analysis lead us to the identification of software error handling, local and global invariants recovery and error treatment as the correct separation of concerns for abnormal event handling. Within our framework, the executable assertions mechanism are responsible for providing error detection and software fault tolerance. The exceptions handling mechanism is responsible for providing error recovery and the normal language constructs are responsible for error treatment.

The separation of concerns together with the classification of error types and the application of the design by contract, provide a clear assignment of responsibilities for each code section and software element for the handling of abnormal situations. Also, it allows a decrease in the code necessary for abnormal event handling and promotes a discipline for design and coding which could be considered as an schema for fault avoidance. The criterions proposed are the basis for a novel exception mechanism proposed for C written embedded systems. We analyzed the differences and advantages of our exceptions mechanism, and compared it against those of other object oriented languages of the C family or based on the Design by Contract as well as with other extensions to the C language.

7. References

[1] David Abrahams, "Exception Safety in STLport," (STLport site, 2001http://www.stlport.org/doc/exception_safety.html.

[2] Eric Allman and David Ben. "An Exception Handler for C", Proc. of the Summer 1985 USENIX Conference, 1985.

[3] Algirdas Avizienis, "The N-Version Approach to Fault-Tolerant Software," IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, pp. 1491-1501, 1985.

[4] Tom Anderson y P. Lee. "Fault Tolerance Principles and Practice". Second Edition. Springer-Verlag, 1990.

[5] Tom Cargill, "Exception Handling: A False Sense of Security", C++ Report. November-December 1994.

[6] Gregory Colvin, "Exception Handling In ANSI C", C/C++ Users Journal. August 1991.

[7] Ingermar J. Cox Narain H. Gehani, "Exception Handling in Robotics", Computer, v.22 n.3, p.43-49, March 1989.

[8] Flaviu Cristian, "Exception Handling and Software Fault Tolerance", Transactions on Computer Systems, vol. C-31, no. 6, pp 531-540, June 1982.

[9] C. Fetzer, K. Hogstedt, P. Felber, "Automatic Detection and Masking of Non-Atomic Exception Handling", Int. Conf. on Dependable Systems and Networks, 2003.

[10] Narain H. Gehani, "Exceptional C or C with Exceptions". Software-Practice and Experience. Vol. 22(10), Oct. 1992.

[11] James Gosling, Bill Joy and Guy Steele. "The Java Language Specification", Addison- Wesley, 1996.

[12] Jim Gray, "Why Do Computer Stop and What Can Be Done About It", Proc. of the 5th Symposium on Reliability in Distributed Software and Database Systems, 1986.

[13] John B. Goodenough, "Exception Handling Issues and Proposed Notations", CACM, 18(2), 683-96. Dec. 1975.

[14] A. Hejlsberg, S. Wiltamuth, P. Golde. "The C# Programming Language", Addison Wesley, 2003.

[15] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," CACM, 12(10):576–583, Oct. 1969.

[16] J.D Ichbiah,et-al. "Rationale for Design of Ada Programming Language". SIGPLAN Notice, 14, 6, 1979.

[17] Jean-Claude Laprie, "Dependability -– Its Attributes, Impairments and Means," in B. Randell, et. al. (eds.), Predictably Dependable Computing Systems, 1995.

[18] P. A. Lee, "Exception handling in C programs". Software-Practice and Experience, 13(5), 389-409. 1983.

[19] Bertrand Meyer, "Object-Oriented Software Construction", Prentice-Hall, 2da Edition, 1997.

[20] Richard Mitchell and Jim McKim, "Design by Contract, by Example", Addison-Wesley, Boston, MA, 2002.

[21] Doug Moen, "A Discipline for Error Handling", Proc. of the Summer '92 USENIX Conference, June 8-12, 1992.

[22] Herald M. Mueller, "Pattern Languages for Handling C++ Resources in an Exception-Safe Way", 2nd USENIX Conference on Object-Oriented Technologies, 1996.

[23] M. Page-Jones, "The Practical Guide to Structured Systems Design," Yourdon Press Comp. Series, Prentice Hall, 1988.

[24] David L. Parnas, "On the criteria to be used in decomposing systems into modules", CACM, Dec 1972.

[25] David. L. Parnas, "The Influence Of Software Structure On Reliability", In Current Trends in Programming Methodology, pp. 111--119. Prentice Hall, April 1977.

[26] B. A. Rafnel, "A transaction approach to error handling", Hewlett-Packard Journal, vol. 44, no. 3, June 1993.

[27] Jack W. Reeves. "Using Exception Effectively: Part I – Coping With Exception", C++ Report, Mar/Apr 1996.

[28] D. Reimer, H. Srinivasan, "Analyzing Exception Usage in Large Java Applications". Workshop on Exception Handling in Object Oriented Systems at ECOOP 2003.

[29] Eric S. Roberts: "Implementing exceptions in C". TR. 40, Systems Research Center, DEC, March 21, 1989

[30] David S. Rosenblum, "A Practical Approach to Programming With Assertions", IEEE Transactions On Software Engineering. Vol. 21, No. 1, January 1995

[31] Bjarne Stroustrup, "The Design and Evolution of C++", Addison-Wesley, Reading, MA, 1990.

[32] Herb Sutter, "When and How to Use Exceptions", C/C++ Users Journal, August, 2004.

[33] Herb. Sutter, "ACID Programming", Sept 1999

[34] Herald Winroth, Matti Rendahl, "Exception Handling in C", C/C++ Users Journal, October, 1993.