15th Triennial World Congress, Barcelona, Spain

# DEALING WITH EXCEPTIONS IN SAFETY-RELATED EMBEDDED SYSTEMS

**Wolfgang A. Halang** [*] and **Matjaž Colnarič** [**]

[*] *Faculty of Electrical and Computer Engineering*
*FernUniversität Hagen, D-58084 Hagen, Germany*
wolfgang.halang@fernuni-hagen.de
[**] *Faculty of Electrical Engineering and Computer Science*
*University of Maribor, SI-2000 Maribor, Slovenia*
colnaric@uni-mb.si

Abstract: In embedded hard real-time systems, tasks must complete their executions within predefined time frames. A necessary pre-condition to achieve this requirement is predictability of their temporal behaviour.
Here, the main focus is on handling exceptions in such systems. When handled in a classical way, they necessarily jeopardise the ultimate requirement, temporal predictability. Hence, it is argued that exceptions must be either prevented or avoided, as far as this is possible.
For the remaining non-preventable and non-avoidable catastrophic exceptions, a technique in form of syntactic means is presented allowing to handle them in a well-structured and predictable way, and as painlessly as possible. The technique is based on recovery blocks with pre- and post-conditions. Finally, a method for the estimation of the resulting temporal behaviour (worst case execution time) is described.

Keywords: Hard real-time systems, embedded systems, high-integrity requirements, safety-related systems, exception handling, real-time programming languages, worst case execution time (WCET) estimation.

## 1. INTRODUCTION

Embedded systems are generally employed in control applications. As a rule, they operate in the hard real-time domain, which means that their integrity does not only depend on the functional correctness of the results, but also on whether these are produced within specified time periods. Depending on the applications, these systems are often safety critical; their malfunction may cause major damage, loss of equipment, or even endangerment of human lives. Thus, for such systems high integrity and safety are required, and mechanisms must be devised to cope with partial or complete failures.

In recent years, the domain of real-time systems substantially gained research interest. Certain subdomains have been examined very thoroughly, such as scheduling and analysis of program execution times. It is typical that most of this research was dedicated to higher level topics assuming that the underlying features behave fully predictably. While even in the systems usually employed in process control, testing of conformance to functional specifications is well established, temporal circumstances are seldom consistently verified. It is almost never proven at design time that such a system will meet its temporal requirements in every situation that it may encounter. Such proof should be provided by a priori schedulability analysis.

For schedulability analysis, worst case execution times (WCET) of tasks must be known in advance. These, however, can only be determined if a system functions predictably. To assure overall predictability, all

the system's layers must behave predictably in the temporal sense, from the processor to the system architecture, language, operating system, and exception handling.

Exception handling is one of the most severe problems to be solved in case temporally predictable behaviour is required. When an exception occurs in a program, the latter is inevitably delayed causing a serious problem with respect to the a priori determined execution time. Therefore, exceptions should be prevented by all means, whenever and wherever it is possible (Black, 1983). If it is not possible to prevent them from occurring, they should be handled in a consistent and safe way in conformity with the design guidelines holding for hard real-time systems, i.e., timeliness, simultaneity, predictability, and dependability (Halang and Stoyenko, 1991). The need for consistent solutions to the exception handling problem is exacerbated by the fact that exceptions often result from critical system states when computer control is needed most.

In this paper, a concept for dealing with exceptional events that may occur in embedded control applications is presented. The main objective is to propose practically usable syntax constructs for exception handling. It is not our intention to analytically explore theoretical background of these matters. Throughout the paper references to much more thorough and basic research results on different topics are given.

First, in Section 2, we commence with some general considerations providing broad conceptual guidelines for the organisation of exception handling in embedded systems. The exceptions occurring in embedded hard real-time systems are classified. It is argued that a number of them can and should be either prevented or avoided by simple measures and means.

In Section 3 it is shown constructively how behavioural predictability can be achieved even in the presence of the remaining catastrophic exceptions. Some known solutions to handle them are summarised, which were combined in the approach implemented. An analysis of the impact is given, which the approach has on the overall predictability of process timing. Finally, some advantages and drawbacks of the proposed approach are evaluated.

## 2. GENERAL CONSIDERATIONS

There are various definitions and classifications of exceptions.

The main idea of traditional exception handling is to trigger the execution of dedicated routines (handlers) when specified events occur. This way, the worst case execution times of the tasks involved are prolonged by those of the exception handlers. Since neither the event occurrences nor just their frequencies can be anticipated, it is impossible to predict the tasks' WCET. This, in turn, prevents schedulability analysis and, thus, predictable time behaviour.

As early as 1983, Black reasons in his PhD thesis (Black, 1983) against exception handling. He strictly distinguishes between exceptions and catastrophes: the difference is not due to severity but to their very nature. To handle exceptions, one has to be aware of what can happen. Once this is the case, exceptions can be handled as ordinary events. Programmers tend to service desired events directly in their programs, and undesired ones as exceptions. It is, however, irrelevant whether an event is desired or not.

Being relevant to hard real-time systems, exceptions occurring when tasks violate their deadlines

A closer examination of the possible sources of exceptions in hard real-time systems leads to the following classification into three categories: preventable, avoidable and catastrophic exceptions.

*Preventable Exceptions.* The best way to handle exceptions is to prevent them from occurring at all — as far as this is possible. This can be achieved by, e.g., imposing certain restrictions on the use of potentially dangerous features such as dynamic data structures, virtual addressing, recursion or dynamic pointers.

Further exceptions can be caused by illegal data, such as illegal operations and/or operands (e.g., square root of a negative argument, array or string index out of range, declaration/run-time value mismatch) or range overflow or underflow in arithmetic operations. They can be prevented by strict type checking in the language used, so that possible irregular operations are detected and reported already at compile time. An example for this approach is the exception-less language NewSpeak (Currie, 1988) supported by an appropriate safe architecture (Kershaw, 1987).

It is a good practice to employ the principle of the IEEE binary floating point arithmetic standard (IEE, 1985; IEC, 1989). The input and output data types are extended by two "irregular" values representing "signed infinity" to accommodate overflows and underflows, and "undefined (not-a-number – NaN)" to formalise the results of invalid operations.

*Avoidable Exceptions.* There are situations in which unusual (or irregular) events cannot be prevented from occurring. In certain cases they can be anticipated during the design phase. In these cases they should be included into the specifications to be handled adequately. Thus, dealing with them becomes part of the application software instead of relying on some universal system exception handling.

*Catastrophic exceptions.* When a completely unexpected and usually undesired event occurs ("the im-

possible happens" (Black, 1983)), for instance due to a hardware failure, a residual software error, or wrong specifications, it should not be considered as an exception, but as a catastrophe. Catastrophes are not to be serviced (in the sense of exceptions), but to be survived by bringing the system into a pre-defined, safe and stable state.

## 3. COPING WITH CATASTROPHIC EXCEPTIONS

The concept of exception handlers assigned to operations can be found in a number of relevant papers and implementations (Goodenough, 1975; Liskov and Snyder, 1979; Stroustrup, 1991; Marlow *et al.*, 1994). In the latter paper it is also shown how the delays caused by exception handlers can be considered in schedulability analysis.

Our approach, however, is based on the reference study in the domain of non-preventable exceptions which was carried out by Cristian throughout a number of years (Cristian, 1982; Cristian, 1984).According to this work, exceptional situations can be dealt with by programmed exception handling, and by default exception handling based on automatic backward or forward recovery using recovery blocks. Since programmed exception handling should be included in the requirements of embedded hard real-time systems and can, thus, be treated as normal actions, in the following the latter approach based on recovery blocks is considered further.

The principle of *backward recovery* is to return to a previous consistent system state after an inconsistency is detected by consistency tests called post-conditions. This can be performed in two ways, viz., (a) by the operating system recording the current context before a program is "run" and restoring it after unsuccessful program termination, or (b) by recovery blocks inside the context of a task whose syntax reads as follows:

RB ≡ **ensure** *post* **by** $P_0$ **else by** $P_1$ **else by** ...
**else** *failure*

where $P_0$, $P_1$, etc. are alternatives which are tried consecutively until either consistency is ensured by meeting the *post*-condition, or the segment *failure* is executed. Each alternative should be independently capable of ensuring consistent results.

In the *forward error recovery technique* it is tried to obtain a consistent state from partly inconsistent data. Which data are usable can be determined by consistency tests, error assumptions, or with the help of independent external sources.

### 3.1 *Syntax and Semantics*

To handle catastrophes, we propose to use a combination of pre-conditions, post-conditions and modified recovery blocks implementing both backward and forward recovery. The syntax of this scheme (based on the syntactic rules of PEARL) is shown in Figure 1.

```
block ::= block_begin block_tail

block_begin ::BEGIN
              | PROCEDURE parameters & attributes;
              | TASK parameters & attributes;
              | parameters REPEAT

block_tail ::=[declaration_sequence]
              [alternative_sequence] END;
declaration_sequence ::= block-specific declarations
              [PRESERVE global_var_list]
alternative_sequence ::=
             {[ALTERNATIVE [PRE bool-exp;]
              [POST bool-exp;]]
              [statement_sequence]}
```

Fig. 1. Syntax of an exception handling mechanism

A block (explicit block, task, procedure, loop, or any other block structure) consists of alternative sequences of statements. Each alternative can have its own pre- and/or post-conditions, represented by Boolean expressions. When the program flow enters a surrounding block, the state variables, that are modifiable by alternatives which might fail, are stacked. Then, the first alternative statement sequence, whose pre-condition (if it exists) is fulfilled, is executed. At the end, its post-condition is checked, and if this is also fulfilled, execution of the block is successfully terminated. If the post-condition is not fulfilled, the next alternative is checked for its pre-condition and eventually executed. If necessary, values of the state variables recorded at the beginning of the block are first restored.

If an alternative fails, any effect on the system state must be discarded. Thus, it is necessary that the original value of any variable, which was modified inside this alternative, is restored. For this purpose, the state of any such variable was stacked at the time of entering the block. Whether and which variables must be stacked can be determined by the compiler. It is only necessary to restore non-local variables that appear on the left-hand side of assignments in alternatives which have post-conditions, since only they may fail after modifying the state. It is a task of the compiler to scan the block for such variables and take care of their stacking. Further, after a non-successful evaluation of a post-condition, only the variables that were modified in this alternative are automatically restored.

Stacking all global variables that can be modified within a block may require a relatively large amount of time. There are situations that the value of a global variable is not needed any more after an unsuccessful termination of an alternative. In such situations the application programmer may wish to declare which modifiable global variables should be restored after

an unsuccessful alternative. This can be requested by the optional PRESERVE declaration in the declaration_sequence. If such a declaration is present, the automatic search for modifiable global variables is overridden. Hence, the explicitly given list must contain the complete set. The compiler then scans for global variables that are both in the list and appear on a left-hand side in the alternative program, and restores their original values after an unsuccessful try.

A good technique which outsources the above problem is to work with private copies of global state variables inside the alternatives that may cause backward recovery, and to export their values after a successful post-condition check. However, this is more time-consuming, especially when there are more such alternatives in a block, which require (counter-productive) transfer of global into local variables and vice versa.

Since embedded hard real-time systems, on which we focus in this paper, are, as a rule, used in process control, a severe problem arises if there are any actions triggered like commencing a peripheral process which cause an irreversible change of initial state inside of an alternative that has failed. In this case, backward recovery is generally not possible. As a consequence, it is our suggestion that no physical control outputs should be generated inside the alternatives which may require backward recovery in case of failure. Then, only forward recovery is possible, bringing the system to a certain pre-defined, safe and stable state.

Both forward and backward recovery methods can be implemented using the proposed syntax. In the following these approaches are shown.

- **Backward recovery:** Bearing in mind the dangers of backward recovery in process control systems, it may be implemented carefully. Backward recovery can be activated by the post-condition an alternative must meet. Its functioning is obvious: if an alternative fails to meet its post-condition, the next alternative fulfilling its pre-condition is used to perform the task of the block. Thus, it is necessary to restore the system state variables possibly modified in previous unsuccessful alternatives.
- **Forward recovery:** This technique may be somewhat less obvious. When an alternative fails as indicated by not meeting its post-condition, it may be followed by one or more alternatives with pre- but without post-conditions. Probably, these alternatives will not completely fulfill the task, since their results are not verified by post-condition checks. More likely, they will incorporate certain fault-tolerance measures. These are alternatives implementing forward recovery: the first one with successfully evaluated pre-condition is executed, and the block is left without further checking. With the pre-conditions it is possible to check the consistency of data and/or system state after the failure of a previous alternative. It is assumed that the situation is resolved by bringing the system into a consistent and safe state in conformance to the specifications. Although the function of the block may have failed (at least to a certain extent), the process execution may be continued safely.

The backward recovery alternatives should contain diversely designed and coded programs to cope with specification errors and to eliminate possible implementation problems or residual software errors. They may employ alternative design solutions or redundant hardware resources, when problems are expected. A further possibility is to assert gradually less restrictive pre- and/or post-conditions and, thus, to degrade performance gracefully in the case of exceptional situations.

By the means presented in (Verber *et al.*, 1996) it is also possible to bound the execution times of alternatives to trap timing errors.

If there is no alternative, whose pre- and post-conditions are fulfilled, the block execution is unsuccessful. If the block is nested inside an alternative on the next higher level, this alternative fails as well (the failure is propagated), and control is given to the next one on this level, thus providing a chance to resolve the problem in a different way. On the highest level, however, the last alternative must not have any pre- or post-conditions. It must solve the problem by applying some safe action like employing robust fault-tolerance measures or performing smooth shut-down. Since the system is then in an extreme and unrecoverable catastrophic condition, different control and timing policies are put into action, requesting safe termination of the process and, possibly, post mortem diagnostics. For the case of extremely safety-critical applications, in this ultimate alternative very basic electrical or mechanical back-up systems can be engaged.

The proposed exception handling mechanism was integrated into a laboratory prototype of the high-level real-time programming language miniPEARL (Verber *et al.*, 1996). A new version featuring object-orientation is currently in preparation.

### 3.2 *Considering Recovery in WCET Analysis*

Using the exception handling mechanism described above, the worst case program execution times required for schedulability analysis can be estimated at compile time. In the following paragraphs three different cases are considered.

*(a) Exclusively backward recovery* (all alternatives have post-conditions): In the worst case execution time estimation all times must be considered, i.e., time for stacking all global variables' contents, for evaluating all pre- and post-conditions, for executing the program alternatives, and times to restore used variables.

$$WCET = t_{st} + \sum_{i=1}^{n} \left( t_{pre_i} + t_{body_i} + t_{post_i} + t_{rest_i} \right)$$

with

| | |
|---|---|
| $n$ | number of alternatives in the block |
| $WCET$ | worst case block execution time |
| $t_{st}$ | time to store global variables |
| $t_{pre_i}$ | i-th alternative pre-condition evaluation time |
| $t_{body_i}$ | i-th alternative program execution time |
| $t_{post_i}$ | i-th alternative post-condition evaluation time |
| $t_{rest_i}$ | time to restore global variables in i-th alternative |

*(b) No backward recovery* (no alternatives have post-conditions): In this case it must be scanned for the maximum time composed of an alternative body execution time plus the sum of unsuccessful pre-condition evaluation times of all preceding alternatives. There is no stacking or restoring of variables.

$$WCET = max_{k=1,n} \left( t_{body_k} + \sum_{i=1}^{k} t_{pre_i} \right)$$

*(c) Mixed alternatives with and without post-conditions:* In this case, the estimation of the worst case execution time is slightly more complicated. During operation, alternatives are tried one after another according to their sequence in the block. Thus, execution times are evaluated as follows:

- the execution time of the sequence of alternatives with post-conditions is calculated as in case (a) and is added to the execution time of the body of the subsequent alternative without post-condition if it exists, or forms a virtual alternative without pre- or post-condition if it stands at the end of the block,
- afterwards, one proceeds as in case (b).

Actually, the last method (c) is generally valid and also applicable in both previous cases.

Especially the backward recovery method inevitably yields pessimistic estimations of execution times. However, this is not due to this specific solution. In safety-critical hard real-time systems it is always necessary to consider worst case execution times, which must also include handling of exceptional situations. Depending on the performance reserve of a system, more or less alternatives may be provided, performing more or less degraded functions. In extremely time-critical systems, just a single alternative in the highest level block may be implemented performing merely a robust fault-tolerant and fail-safe measure, like safe and smooth shut-down.

To cope with the problem of pessimism in run-time estimation of the execution of alternatives, some further solutions are possible. As an example, each subsequent alternative of a set of backward recovery alternatives may be bounded to half of the execution time of the previous one. Thus, the block will terminate in at most twice the execution time of the primary alternative.

Further, from a failure of an alternative it is possible to deduce which subsequent alternatives in subsequent blocks are reasonable and which are not, and to set their pre-conditions accordingly. However, this requires a more sophisticated run-time analyser.

## 4. CONCLUSION

Exception handling presents one of the most severe obstacles to the predictability of the temporal behaviour of hard real-time systems. In this paper, the possible exceptions were classified. The first group comprises exceptions whose occurrences can be prevented by appropriate measures. The exceptions of the second group can be avoided at run-time. Their servicing is implicit, and the corresponding service times can be predicted by common methods of execution time estimation. To cope with the third group, i.e., non-preventable and non-avoidable exceptions, a well-structured method was introduced, providing sequences of gradually more and more evasive software reactions with fully predictable execution behaviour.

Embedded hard real-time systems for process control often operate in safety-critical environments. Uncontrolled malfunctions can have severe consequences with regard to repair costs, production loss, or even endangerment of human health or lives. By our approach, overall system safety is greatly enhanced.

Possible system failures are already being considered during the specification phase. During the design phase, alternative solutions are devised and prepared. Having to consider these measures in worst case execution time estimation, performance may be reduced, but safety is retained, since they will either solve the problems, or bring the system into some controlled and safe state. These alternative measures either employ software approaches or redundant hardware means. They are gradually less complex and, thus, less sensitive to disturbances and failures. Therefore, they rely on very simple means of fault-tolerance, employing minimum resources. They may even employ basic electrical or mechanical means.

Applications designed this way fulfill the requirements of hard real-time systems, viz., timeliness, simultaneity, predictability, and dependability. Although worst case analysis necessarily introduces pessimism into run-time estimation, the proposed methodology is usable in practice to develop safety-critical embedded hard real-time applications if the alternative solutions to the critical parts of control tasks are designed reasonably.

## 5. REFERENCES


Black, Andrew P. (1983). Exception handling: The case against. Technical Report TR 82-01-02. Department Of Computer Science, University of Washington. (originally submitted as a PhD thesis, University of Oxford, January 1982).

Colnarič, Matjaž and Domen Verber (2001). Dealing with tasking overload in object oriented real-time applications design. In: *Sixth International Workshop on Object-oriented Real-time Dependable Systems WORDS2001*. Rome, Italy. pp. 226–232.

Cristian, Flaviu (1982). Exception handling and software fault tolerance. *IEEE Transactions on Computers* **31**(6), 531–540.

Cristian, Flaviu (1984). Correct and robust programs. *IEEE Transactions on Software Engineering* **10**(2), 163–174.

Currie, Ian F. (1988). NewSpeak: a reliable programming language. In: *High-integrity Software*. pp. 122–158. Pitman Publishing. London.

Goodenough, John. B. (1975). Exception handling: Issues and a proposed notation. *Communication of the ACM* **18**(12), 683–696.

Halang, Wolfgang. A. and Alexander D. Stoyenko (1991). *Constructing Predictable Real Time Systems*. Kluwer Academic Publishers. Boston–Dordrecht–London.

IEC (1989). *Binary Floating-Point Arithmetic for Microprocessor Systems.* IEC 559:1989.

IEE (1985). *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754-1985.

Kershaw, John (1987). The VIPER microprocessor. Technical Report 87014. Royal Signals And Radar Establishment. Malvern,Worcs, London: Her Majesties' Stationery Office.

Liskov, Barbara H. and Alan Snyder (1979). Exception handling in CLU. *IEEE Transactions on Software Engineering* **5**(6), 546–558.

Marlow, Thomas J., Alexander D. Stoyenko, Stephen P. Masticola and Lonnie R. Welch (1994). Schedulability–analyzable exception handling for fault–tolerant real–time languages. *Real-Time Systems* **7**(2), 183–212.

Stroustrup, Bjarne (1991). *The C++ Programming Language*. Addison–Wesley.

Verber, Domen, Matjaž Colnarič and Wolfgang A. Halang (1996). Programming and time analysis of hard real-time applications. *Control Engineering Practice* **4**(10), 1427–1434.