Supporting High Integrity and Behavioural Predictability of Hard Real-Time Systems

M. Colnarič and D. Verber
University of Maribor, Faculty of Technical Sciences
Smetanova 17, Maribor, Slovenia
colnaric@uni-mb.si
AND
W. A. Halang
FernUniversität Hagen, Faculty of Electrical Engineering
D-58084 Hagen, Germany
wolfgang.halang@fernuni-hagen.de

Keywords: hard real-time systems, high-integrity requirements (safety-related systems), exception handling, real-time programming languages, process run-time estimation

Edited by: Marcin Paprzycki and Janusz Zalewski
Received: February 19, 1994 **Revised:** October 30, 1994 **Accepted:** December 19, 1994

The main objective of this paper is to present a method for handling non-preventable and non-avoidable catastrophic exceptions in embedded hard real-time environments in a well-structured and predictable way, and as painlessly as possible.
First, apt hardware and software platforms which are pre-requisite for predictable system behaviour are briefly presented. Then, some existing techniques are shown and their suitability for implementation in embedded hard real-time environments is discussed. Further, a classification of exceptions and our own approach for handling them is presented and elaborated. Finally, a method for the estimation of the resulting temporal behaviour is described.

1 Introduction

In this paper, embedded hard real-time systems are dealt with. In general, they are employed to control different processes; the integrity of these applications relies on their temporally and functionally correct operation. Depending on the application, these systems can be extremely safety critical; their malfunction may cause major damage, material loss, or even endangerment of human lives. Thus, for such systems high integrity and safety is required, and mechanisms must be devised to cope with partial or complete failures.

While in the systems, which are usually used in process control, testing of conformance with 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.

In his reference paper [20], Stankovic is unmasking several misconceptions in the domain of hard real-time systems. Seemingly the most characteristic one is that real-time computing is often considered fast computing. It is obvious that computer speed itself cannot guarantee that specified timing requirements will be met.

Instead, a different ultimate objective was set: predictability of temporal behaviour. Being able to assure that a process will be serviced within a predefined time frame is of utmost importance. In multiprogramming environments this condition can be expressed as schedulability: the ability to find a schedule such that each task will meet its deadline [22].

For schedulability analysis, execution times of tasks must be known in advance. These, however, can only be determined if a system functions predictably. To assure overall predictability, all system layers must behave predictably in the temporal sense, from the processor to the system architecture, language, operating system, and exception handling (layer-by-layer predictability, [21]).

In recent years, the domain of real-time systems substantially gained research interest. Certain sub-domains have been examined very thoroughly, such as scheduling and analysis of program execution times. It is typical that most of the research done was dedicated to higher level topics and presumes that the underlying features behave fully predictably.

Exception handling is one of the most severe problems to be solved when a system is to behave predictably. By an exception any unexpected intrusion into the normal program flow which cannot be considered during schedulability analysis phase is meant. It is usually related to residual specification and implementation errors and to failures. Anticipated timing events and events from the environment, which trigger associated processes, do not belong to this category. They should be implemented in a way, which does not cause any non-deterministic delays in the execution of the running tasks. That can be achieved by migrating event recognition and operating system services out of main task processors [11], and was also implemented in the Spring project [19]. Results of our previous studies were presented in [4] and are used in the design of an experimental platform as described in the next section.

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 [1]. If it is not possible to prevent them to happen, they should be handled in a consistent and safe way in conformity with the hard real-time systems design guidelines, i.e. timeliness, simultaneity, predictability, and dependability [13]. The need for consistent solutions to the exception problem is exacerbated by the fact that exceptions are often the results of some critical systems states, which is when computer control is needed most.

In this paper we show constructively how behavioural predictability can be achieved by presenting an experimental system and considering different aspects of its design. Although the main emphasis of the paper is on consistent exception handling, it is necessary to present some principles used to provide the necessary pre-conditions for deterministic system behaviour; only then it is reasonable to consider the upper layers of a system design. In Section 2 we start off with describing the basic layers of an asymmetrical parallel hardware architecture and the operating system concepts which prevent process control tasks to be disturbed (and thus delayed) by events occurring in the environment. Further, in Section 3, a real-time programming tool supporting the architecture is described by which process control programs with deterministic temporal behaviour can be designed and their run-times determined.

Exception handling was integrated into a high-level programming language, which is the subject of Section 4. First, we classify exceptions and show that a number of them can be either prevented or avoided. Further, we summarise some known solutions to handle the remaining exceptions, which were all combined in the implemented approach. Finally, an analysis of the impact the approach has on overall process timing predictability is given.

2 Concept of an Experimental Hardware Platform

In multi-tasking systems, dynamic scheduling algorithms to generate appropriate schedules must be implemented. The ones which fulfill the requirement that all tasks must meet their deadlines are referred to as feasible. In the literature, several such algorithms have been reported (an overview is given in [13]). For our purpose, the earliest-deadline-first scheduling algorithm is chosen. It has been shown that it is feasible for scheduling tasks on single processor systems; with the throw-forward extension it is also feasible on homogeneous multiprocessor systems. However, this extension leads to more pre-emptions and is more complex and, thus, less practical.

For process control applications, where process interfaces are usually physically hard-wired to sensors and actuators establishing the contact to the environment, it is natural to implement either single processor systems or dedicated multiprocessors acting and being programmed as separate units. Thus, the earliest-deadline-first sche-

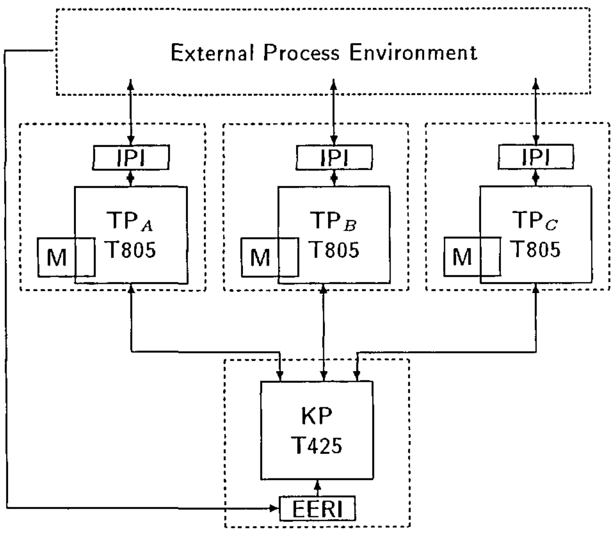

Figure 1: Scheme of an experimental hardware platform

duling policy can be employed without causing any restrictions, resulting in a number of advantages discussed by Halang and Stoyenko [13].

In the classical computer architecture the operating system is running on the same processor(s) as the application software. In response to any occurring event, the context is switched, system services are performed, and scheduling is done. Although it is very likely that the same process will be resumed, a lot of performance is wasted by superfluous overhead. This suggests to employ a second, parallel processor, to carry out the operating system services. Such an asymmetrical architecture turns out to be advantageous, since, by dedicating a special-purpose, multi-layer processor to the real-time operating system kernel, the user task processor(s) are relieved from any administrative overhead.

This concept was in detail elaborated in [11] and further refined in [3, 4]. Our experimental hardware platform is to a high extent complying with these principles, and is currently under construction. In Figure 1 it is shown that it consists of task processors (TPs) with inteligent process interfaces (IPI) and a kernel processor (KP) with an external event recognition interface (EERI), which are fully separated from each other.

The external process is controlled by tasks running in task processors without being interrupted by the operating system functions. Any event from the environment is fed to the kernel processor and scheduling is performed based on the modified earliest-deadline-first policy: the intention is to find a schedule such that all waiting tasks including the newly arrived one meet their deadline while the running task remains in execution. Thus, a running task is only pre-empted if it is necessary to assign the highest priority to an incoming task in order to allow that all tasks meet their deadlines.

The task processors are implemented with INMOS T805 transputers. In the task processors' external memory the code of each task assigned to be run is loaded. Also, a part of the control blocks of these tasks is residing there, holding the context of eventually pre-empted tasks. The fast on-chip RAM of the transputers is holding task internal variables except for the large data structures which are held in the external memory.

The IPI process interface is based on a Motorola MC68000 microprocessor which adds the necessary intelligence to peripheral devices. It is accessible by a bi-directional link via an INMOS converter and is acting as a slave to the task processor(s). Services of the intelligent process interface are available by calling pre-defined peripheral device drivers and providing parameters and data.

Synchronisation of tasks running in different task processors is carried out with the help of semaphores residing in the kernel processor and being accessible through systems calls.

The kernel processor is responsible for all operating system services. It consists of an INMOS T425 transputer performing the operating system kernel services, and a Motorola MC68000 based external event recognition interface. The latter's task is administering the real-time clock in the form of Julian time, receiving signals from the process environment, providing them with time stamps, and periodically triggering events by sending messages to the transputer containing information about all events that happened recently, and serving as a synchronisation means.

The time between two synchronisation messages from the EERI is further sub-divided in slots in the kernel processor. In these slots the information from the external event recognition module is processed, time events are administered, and OS service calls from the task processors are serviced, each triggering scheduling of an associated application task.

It is to be mentioned that our nomenclature is not strictly conforming with [11], although the functions implemented in our architecture comply with the layers proposed there. The external event administration part of the hardware layer functions is implemented in the EERI, the others — primary and secondary reaction level — in the kernel processor. The hierarchy is retained by executiong these functions in strictly defined slots. We are considering migrating the scheduling-related secondary reaction level services into a separate transputer to enhance performance.

Through this concept, preventing nondeterministic interruptions from the environment, careful avoidance of the sources of unpredictable processor and system behaviour, loose coupling of task processors, and synchronous operation of the kernel processor, the predictability of the temporal system behaviour will provide the necessary basis for the higher system design levels.

3 Concept of a Real-Time Programming Tool

To program applications on the above hardware platform a tool is being constructed, in which the proposed exception handling mechanism is built in. Its ultimate objective is to produce temporally predictable and optimal program code for embedded hard real-time applications, and estimations of their execution times.

In the tool two parts are closely integrated: a compiler for an adapted standard real-time programming language, and a program execution time analyser. The latter is providing the necessary information for a schedulability analyser which is currently beyond the scope of our research.

In the design of the tool, the following guidelines were followed:

1. *Target system independence.* The compiler should produce executable application code for a variety of target systems. This is achievable by implementing target system specific macros which transpose each element of intermediate code into a corresponding piece of executable code. In the specification file for each target system its own set of macros is defined.

2. *Generation of efficient code.* Although being system independent, the compiler is expected to generate fast and compact code. This can be achieved by the simplicity of the programming language, and the possibility of global syntax tree optimisation (register scheme, local and global variables' locations and other implementation specific information are given in the system specification file). Also, in translation macros full information about the operands (constant, register, local or global variables) is contained.

3. *Realistic estimation of task execution times.* A drawback of many methods for task execution time estimation is that they yield such pessimistic results that their relevance is seriously diminished [18]. To cope with that, the tool supports two different methods to determine the execution time of a task: compile-time program analysis and direct measurement of worst-case (partial) task execution time.

3.1 MiniPEARL

To program an application, the programming language miniPEARL is introduced. It is a simplified version of PEARL [9], a standard language for programming real-time applications, which, however, may produce temporally unpredictable code for several reasons. To eliminate these problems, PEARL's syntax is modified. Further, to support efficient mapping onto typical target architectures certain features are reduced. Finally, it is enhanced by some constructs specific to real-time systems, proposed by Halang and Stoyenko [13, 12]. MiniPEARL is described in more detail in [23].

The main differences between PEARL and miniPEARL are:

1.*There are no GOTOs.* The use of GOTO statements can result in unstructured and hardly manageable code. Instead of these, EXIT and LOOP statements are introduced for preliminary exit from an innermost structure, and for immediate initiation of the next iteration of a loop, respectively.

2.*Each loop block is strictly bounded.* In the REPEAT statement, lower and upper counts of a loop are obligatory and defined with compile-time constant expressions to limit the number of iterations.

3.*Pointers and recursion are not allowed.* Dynamic data structures and recursion can result in severe memory management problems. They may cause temporally non-deterministic actions that cannot be considered in timing analysis.

4.*Signals are not directly supported.* In our architecture model interrupts and signals are managed by the kernel processor. Events can be induced by the synchronisation mechanism.

5.*Each statement execution is temporally bounded.* Commands whose execution time is non-deterministic must be either forbidden, or taken special care of in a real-time systems. Each of such commands which are unavoidable must be temporally guarded, and time-out alternatives must be defined explicitly. Commands that must be guarded are the ones that are dealing with process inputs and outputs (if handshaking is implemented), and synchronisation mechanisms.

6.*Explicitly asserted execution time.* Frequently, because of the nature of a program, estimation may yield very pessimistic execution times. To resolve this problem, additional information about program execution must be given by the programmer. This can be done by adding new constructs (pragmas) into program code as proposed in [18]. But such constructs require complex analysis and are not feasible for all situations. To overcome this problem, the execution time of code segments known through competent measurement, detailed analysis of the program behaviour, experience, reuse, etc. may be explicitly asserted by the system developer who also takes the responsibility. In such case, execution time analysis is overriden. However, to guarantee that the actual execution time will not be longer than declared, blocks must be guarded by time-out controls, and time-out action must be present.

8.*DATIONs are not used.* Mass storage and asynchronous input/output devices as used in PEARL are not suitable for hard real-time systems. For this reason and because of the relative complexity of these structures, DATIONs are excluded from the structure of the language. Input/output devices (registers) are accessed at the lower programming level.

9.*Improved task activation scheme.* In miniPEARL task activation, deactivation, etc. can be done through signals from the environment, time-related conditions, or specific states of synchronisers, as proposed in [13]. A time-related and one non-time-related condition may be combined.

10.*Scheduler support.* The scheduling algorithm performed in the kernel processor relies on the residual execution time of a task. This time is computed as maximum execution time of the task minus cumulative running time. However, the actual execution time is expected to be shorter than the estimated one. To achieve better performance, the actual residual task execution time can be explicitly asserted at ceratin points to update the estimated one.

3.2 Estimation of Task Execution Times

To allow for schedulability analysis, precise execution times of application tasks must be known in advance. In our tool, two methods for the estimation of program run-times are supported:

1. *Analysis of executable code.* In this method, an automatic analyser is used to estimate execution times (compare also [18, 17]). Source code is transformed into an intermediate form (modified syntax tree) prior to executable code generation. Each element of this form is associated with a macro block that is used for two purposes. The first is to generate the code and the second is to obtain its execution time. Because the execution time of the same block can be data-dependent, as much information as possible about operands should be passed to it. The operand can be a register, a constant, a local variable or a global variable. When the macro is expanded, the sum of times needed for accessing these operands is added to the basic execution time of the macro.

2. *Direct measurement of executable code.* This method can be used when more precise execution time than estimated is desired. To achieve that, object code is executed on the target system and the execution time is recorded. Direct implementation of this method has some disadvantages:

– The complete target system must be implemented. That is inappropriate in early phases of development when the target-system is not completely implemented, yet.

– Through recording, only average execution times can be obtained. For usable analysis, however, worst-case execution times are needed. A test scenario to obtain that situation is usually difficult to determine.

– The input/output devices must be active and interact with the environment. Thus, the embedding environment or a simulation of it is needed.

By our approach, these disadvantages are eliminated. Only a task processor or its equivalent must be implemented. The longest path through a task is determined by the compiler and a pilot code is generated running only through that path. From a set of alternative constructs (IF and CASE statements, for example), the longest one is statically routed. All time-guarded commands and input/output variable accesses are replaced by appropriate delays. This pilot code is then executed on the hardware platform or, because of the substitution of every system-specific function, a delay is inserted.

4 Handling of Exceptions in Hard Real-Time Systems

Our previous work in the domain of dealing with exceptions was published in [5]. With the goal to avoid non-deterministic delays in the execution of application tasks it was shown that a great number of exceptions can be either prevented from happening, or they can be handled within the context of task requirements:

– *Preventable exceptions:* Some exceptions can be prevented by restricting the use of potentially dangerous features. Compliance with these restrictions must be checked by the compiler. For example, no dynamic features like recursion, references, virtual addressing, or dynamic file names and other parameters etc. are allowed.

Other features are, e.g., strong type checking (see [8]), or extensions of the input and output data types by two "irregular" values representing "signed infinity" to accommodate overflows and underflows and "undefined", as proposed in the IEEE 32-bit floating point standard [2] implemented also in the INMOS transputers' Floating Point Unit (FPU) (compare also [16]). Thus, computed irregular values do not raise exceptions, but are propagated to the subsequent or higher-level blocks, which must be able to handle them.

– *Non-preventable, anticipated exceptions:* If the potential danger of irregularity can be recognised during design time, it has to be taken care of in the specifications. For example, peripheral devices shall be intelligent, fault-tolerant and self-checking in order to be able to recognise their own malfunctions, and to react in a predefined way if a value which is sent to them is irregular. Further, a number of exceptions resulting from irregular data can be avoided by prophylactic run-time checks before entering critical operations. Many tasking errors are also avoidable by previously using monadic operations to check the system state.

Falling into this category, an obvious and frequently used way of avoiding critical failures in hard real-time systems design is redundancy (an example for consistent implementation of redundancy is the MARS system [15]). Redundant system components must be implemented according to thorough analysis of fault hypotheses.

If there is no way to predict an error, an exceptional situation caused must be handled in order to survive it. These are situations when "the impossible happens" [1], in which programs do not follow their specifications due to hardware failures, residual software errors, or wrong specifications. For example, failure of a part of memory can result in the change of constant values; an error in file management or on a disk is usually unexpected. In safety-critical control systems non-anticipated exceptions may have catastrophic consequences. There it is especially important to implement a mechanism for their safe and consistent handling.

In his early paper, Goodenough [10] presented the idea of assigning default or programmed exception handlers to every potentially dangerous operation. According to the severity of an exception raised the running process was either terminated, or suspended and resumed later. A similar mechanism although considerably more elaborate and adapted for use in hard real-time systems was implemented in Real-Time Euclid [14]. There, exception handlers were (optionally) located within block constructs and were executed in the case of an exception. If there were no exceptions the handlers had no effect except for their impact on a block's execution time estimated by a schedulability analyser, thus making it more difficult to be scheduled. Exceptions may be raised by kill, terminate or except statements, to terminate a process entirely or only its frame, or to execute the handler without termination of the process, respectively.

A reference study in the domain of non-preventable exceptions was done by Cristian [6, 7]. Certain principles from this work were further detailed in [1] and were also adopted in our exception handling mechanism.

According to Cristian, exceptional situations can be handled (a) by programmed exception handling and (b) by default exception handling based on automatic backward recovery using recovery blocks. Since in embedded hard real-time systems programmed exception handling should be included in the system requirements and can, thus, be treated as normal actions, in the following the alternative technique will be dealt with briefly.

The principle of *backward recovery* is to return to the previous consistent system state after an inconsistency is detected by consistency tests called post-conditions. It can be done in two ways, (a) by the operating system recording the current context before the program is "run" and restoring it after its unsuccessful termination, or (b) by recovery blocks inside the context of a task whose syntax is 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 *failure* is executed. Each alternative should be independently capable to ensure 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 presumptions, or with the help of independent external sources.

To handle catastrophes we propose a combination of pre-conditions, post-conditions and modified recovery blocks implementing both backward and forward recovery. Its syntax is shown in Figure 2.

A block (plain block structure, task, procedure, loop, or 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 (see below).

```
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]}
```

Figure 2: Syntax of an exception handling mechanism

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 should be discarded; thus, it is necessary that the original value of any variable is restored, which was modified and lies outside of the scope of the failed alternative. For that purpose, the state of any such variable must be 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 an assignment 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 where 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 the unsuccessful alternative. This can be requested by the optional PRESERVE declaration in the declaration_sequence. If this declaration is present, the automatic search for modifiable global variables is prevented; 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 prevents 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 back.

Since embedded hard real-time systems, which are the main subject of 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 causes an irreversible change of initial state inside an alternative that 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 cause backward recovery in case of failure, i.e., inside those which have post-conditions. In this case only forward recovery is possible, bringing the system to a certain predefined, safe, and stable state.

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

- **Backward recovery:** bearing in mind the dangers of backward recovery in process control systems, it may be (carefully) implemented. Backward recovery can be recognised by the post-conditions 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 do 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. Consider the case where an alternative is checking the success of its operation, according to its design specifications. According to the results of the check, different actions may be taken to resolve different situations. To control the program flow, this alternative then, according to the outcome of these checks, sets some states with which the pre-conditions of alternatives in the subsequent block are set. There may be an alternative with empty statement_sequence whose pre-condition is met if a previous alternative was successful; by this example, classical exception handling can be implemented.

The alternatives should contain independently designed and coded programs to comply with specifications and to eliminate possible implementation problems or residual software errors. They can contain alternative design solutions or redundant resources, when problems are expected. A further possibility is to assert less restrictive pre- and/or post-conditions and to degrade performance gracefully. By the means presented in [23] it is also possible to bound the execution times of alternatives. If one of them fails to complete inside a predefined period, a less demanding alternative is taken.

If there is no alternative, whose pre- and post-conditions are fulfilled, the block execution was unsuccessful. If the block was nested inside an alternative on the next higher level, this alternative fails as well and the control is given to the next one, thus providing a chance to resolve the problem in a different way. On the highest level, the last alternative must not have any pre- or post-conditions. It must solve the problem by applying some conventional actions like employing fault-tolerance measures or performing smooth power-down. Since the system is then in an extreme and unrecoverable catastrophic condition, different control and timing policies are put in action, requesting safe termination of the process and possibly post-mortem diagnostics.

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

(a) Exclusive 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, alternative program execution times, and times to restore used variables.

$$t_{wc} = t_{st} + \sum_{i=1}^{n} t_{pre_i} + t_{body_i} + t_{post_i} + t_{rest_i}$$

where

i	number of alternatives in the block
t_{wc}	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 non-successful pre-condition evaluation times of all preceeding alternatives; there is no stacking or restoring of variables.

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

(c) Mixed alternatives with and without post-conditions: in this case, 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 is 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 execution time estimations. However, this is not due to this specific solution. In safety-critical hard real-time systems it is necessary to consider worst-case execution times, which must also include exceptional conditions. 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 only performing a safe and smooth power-down.

To cope with the problem of the pessimism of run-time estimation of execution of alternatives some further solutions are possible. 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. Also, 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 preconditions accordingly. However, this requires a sophisticated run-time analyser.

5 Conclusion

In order to assure a predictable behaviour of real-time systems, it is necessary to determine a priori bounds for the task execution times. In this paper a consistent design of a computing system for embedded applications operating in the domain of hard real-time is described. While the experimental hardware platform and program development tool are only outlined, the exception handling mechanism, which represents the most severe obstacle to overall predictability, is dealt with in more detail. Catastrophic exceptions are coped with in a well-structured environment by providing sequences of gradually more and more evasive software reactions.

Embedded hard real-time systems for process control often operate in safety critical environments. Uncontrolled malfunctions can have drastic 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 design phase, and alternative solutions are devised and prepared. Having to use them, performance may be reduced, but safety is retained, since they will either solve the problem, or bring the system into some controlled and safe state. These alternative measures either employ software approaches or redundant hardware means, or are gradually less complex and, thus, less sensitive to disturbances and failures. Therefore, they rely on very simple fault-tolerance measures, employing minimum resources. They may even employ electrical or mechanical means, such as safe passive state of inactive relays or automatic activation of mechanical brakes when the system loses control, etc.

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

References

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

[2] W.J. Cody, J.T. Coonen, D.M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F.N. Bis, and D. Stevenson. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro*, 4(4):86–100, August 1984.

[3] Matjaž Colnarič. *Predictability of Temporal Behaviour of Hard Real-Time Systems.* PhD thesis, University of Maribor, June 1992.

[4] Matjaž Colnarič and Wolfgang A. Halang. Architectural support for predictability in hard real-time systems. *Control Engineering Practice*, 1(1):51–59, February 1993. ISSN 0967–0661.

[5] Matjaž Colnarič and Wolfgang A. Halang. Exception handling and predictability in hard real-time systems. In *Proceedings of the 12th International Conference on Computer Safety, Reliability and Security SAFECOMP '93*, pages 371–378, Poznań – Kiekrz, Poland, October 1993. Springer–Verlag, London, 1993.

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

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

[8] Ian F. Currie. NewSpeak: a reliable programming language. In *High-integrity Software*, pages 122–158. Pitman Publishing, London, 1988.

[9] *DIN 66 253: Programmiersprache PEARL, Teil 1 Basic PEARL.* Berlin, 1981.

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

[11] Wolfgang A. Halang. Definition of an auxiliary processor dedicated to real–time operating system kernels. Technical Report UILU-ENG-88-2228 CSG-87, University of Illinois at Urbana Champaign, 1988.

[12] Wolfgang A. Halang and Alexander D. Stoyenko. Comparative evaluation of high–level real–time programming languages. *Real-Time Systems*, 2(4):365–382, 1990.

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

[14] Eugene Kligerman and Alexander Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on*

Software Engineering, 12(9):941–949, September 1986.

[15] Hermann Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, 9(1):25–40, February 1989.

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

[17] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[18] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989.

[19] Krithi Ramamritham and John A. Stankovic. Overview of the SPRING project. *Real-Time Systems Newsletter*, 5(1):79–87, Winter 1989.

[20] John A. Stankovic. Misconceptions about real–time computing. *IEEE Computer*, 21(10):10–19, October 1988.

[21] John A. Stankovic and Krithi Ramamritham. Editorial: What is predictability for real–time systems. *Real-Time Systems*, 2(4):246–254, November 1990.

[22] Alexander Stoyenko. *A Real-Time Language With A Schedulability Analyzer.* PhD thesis, University of Toronto, December 1987.

[23] Domen Verber and Matjaž Colnarič. A tool for estimation of real-time process execution times. In *Proceedings of Software Engineering for Real-Time Applications Workshop*, pages 166–171, Cirencester, September 1993. IEE.