A Simple and Cost-Effective Method to Construct Reliable Real-Time Programs

Received 4th May 2009; accepted 11th September 2009.

Peter F. Elzer, Martin Gollub, Sven Trenkel

Abstract:

This paper presents a proposal for structuring real-time programs in a way that improves their reliability. Basically, it consists of five constructs that have been designed after the model of classical "structured programming", together with a generalized way of dealing with resources. After some considerations about reliability these constructs are presented and explained. Then, a test implementation for two different programming languages and operating systems is briefly described. Finally, some pieces of code give an impression of the character of the method.

Keywords: *real-time systems, real-time programming, structured programming, reliability, graphical repre-sentation.*

1. Introduction

In the dawn of the upcoming era of mechatronics, research concerning real-time systems is becoming a central issue of computer science. A broad variety of problems is discovered that need to be solved. Fortunately, real-time systems have already been the subject of intensive research for decades in the context of electrical and process engineering, and many useful research results as well as applicable engineering traditions have been accumulated.

One area has been of paramount importance in all the discussions over the years: reliability of real-time systems. Of course, there is a great number of aspects that have to be taken into account during the construction of systems with this desirable property. System structure, hardware quality, design methods, simulation, programming languages and guidelines, algorithms and operating systems are just the most important ones. In his teaching, the first author has tried to do justice to as many of these as possible in order to make students aware of the fact that all of them have to be taken into account if a resulting system is to deserve the attribute "reliable".

One part of this effort was in 2004 a joint student project [1]. In this project, a simple and easy to learn method for structured design of reliable real-time systems has been implemented and applied for the first time. Its theoretical foundations have been described before at several occasions [2], [3], [4], [5]. In contrast to the usual approaches to (formal) specification, it is conceptually simple and easy to use. In particular, it is independent of existing programming languages and operating systems. Besides, it can be implemented with relatively little effort and is - after all - directly related to the real-time domain.

In the following two sections the theoretical foundations of the method are briefly presented. Then, the test implementation for the programming languages PEARL [6], [7] and "Real-Time C" [8] is described and, finally, its application is illustrated by means of programming examples.

2. Underlying Design Criteria

2.1. General Considerations

As mentioned above, much work has been done over the past decades with respect to reliability of software (and related properties). As an example (and because it brings some structure into the variety of aspects related to this topic), the work of Laprie [9] shall be used here.

According to a taxonomy he proposes, "reliability" is one attribute (out of six) of a more general property of technical systems: "dependability". He defines it as: "the ability to provide continuity of service". Dependability, in turn, has been defined as: "the extent to which the system can be relied upon to perform exclusively and correctly the system task(s) under defined operational and environmental conditions over a defined period of time, or at a given instant of time" [10].

The other five attributes of dependability are: availability, safety, confidentiality, integrity and maintainability. However, in order to achieve safety, confidentiality and integrity, it is necessary to apply more means than those that are provided by software technology (in a strict sense). But of the remaining three (of the above-mentioned six) attributes, reliability and maintainablity are well within the reach of software technology. So, what can be done about them? Following Laprie's line of thought, reliability is usually impaired by system failures, resulting from errors which, in turn, are consequences of faults. Having stated this, after some elaborations about fault classes and their respective properties and causes, he arrives at the following statement: "Software, and thus design faults, are generally recognized as being the current bottleneck for dependability in critical applications, be they money- or life-critical. A simple reason is that the computer systems involved in such applications are tolerant to physical faults".

The last sentence is particularly interesting insofar as it expresses the observation that - in contrast to the situation some decades ago - in the meantime hardware has obviously reached a satisfactory state.

Design engineers, however, now want to know what can be done to avoid such software design faults. For this purpose, the following means are recommended:

a) fault prevention,

b) fault tolerance,
c) fault removal,
d) fault forecasting.

2.2. Possible Benefits of the Proposed Method

From such considerations Laprie basically derives the necessity of extending complete existing software development models in such a way that they provide such means and thus are appropriate for the development of reliable software systems. However, such an approach includes a great number of individual measures and therefore has not been chosen by the authors.

The presented method is merely intended as a contribution to the reduction of some of the most pressing problems connected with aspects of parallelism in programs written in procedural languages. One of them is the observation that even program developers, who are very competent as far as "classical" algorithms (e.g., in numerical mathematics or numeric control) are concerned, run into difficulties when they have to solve problems of coordination of parallel processes using the usually available means (e.g., free manipulation of processes, signals, semaphores, etc.).

It appears, therefore, obvious to separate these tasks from the algorithmic parts of a program, and to provide program developers with higher level constructs for a number of frequent cases that are easy to understand and fit for easily useable tools like, e.g., macroprocessors or interactive graphical tools. This means that the "descriptional level" has to be above that of usual programming languages. In particular, their functionality concerning coordination of processes or handling of resources should not replace the respective (traditional) mechanisms provided by existing operating systems, but rather build upon and use these. Finally, they have to be conceptually simple and suitable for a graphical notation.

From such an approach the following contributions to the abovementioned means can be expected:

a) **Fault prevention** will be **enhanced by** a **reduction of the number of programming errors**, because:
   a1) **less Lines Of Source Code (LOSC**) should have **to be written** (and the number of faults is basically proportional to the number of LOSC),
   a2) the **complexity of programs is reduced** (again the number of errors in a piece of software increases with increasing complexity), and
   a3) there is a **reduction of design complexity by the separability of development tasks** (specialists normally make less errors in their domain than "allrounders").
b) **Fault tolerance** can be supported by mechanisms for **error detection** and **error recovery** in the software components. In the first place, this can be achieved by means of a comprehensive mechanism for exception handling.

Derived from experience in industry, an additional requirement is the following: due to the long life-expectancy of modern embedded systems, any method should be able to survive several generations of software systems. Usually, it is attempted to achieve this goal by means of standardization. However, experience has shown that it is not possible to standardize any component of or tool for the development of computer systems over more than a few years or outside a limited area of applications. Therefore, the proposed method has to be of a sufficiently high level and - as already mentioned above - independent of any programming language, operating system or computer hardware.

It turned out that a generalization of the principles of "structured programming" fulfils these criteria. For reasons of completeness the basic concepts of that technique shall be briefly presented here.

2.3. "Classic" Structured Programming

One of the real breakthroughs in classical programming was the development of "structured programming" (as, e.g., advocated for by Parnas [11]) in the 1960ies and early 1970ies. One of the most important results of the respective research was the discovery that the control flow of even the most complex sequential programs can - in principle - be described by very few basic constructs. One of the most popular and useful graphical representations of these constructs are the diagrams that have been proposed by Nassi and Shneiderman in the early 1970ies [12]. Basically they visualize the fact that any sequential program can be represented by a linear sequence of "structure blocks", each of which represents one of these constructs.

One of various representations of these diagrams is shown in Fig. 1. As structure blocks it uses the code-sequence, the repetition of a code-sequence under a certain condition, the execution of a number of code-sequences depending on certain conditions, and the alternative execution of two code-sequences.

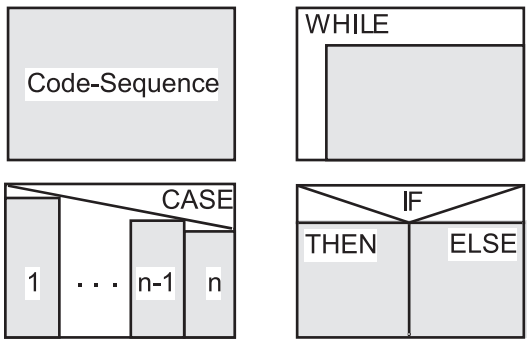

Fig. 1. Nassi-Shneiderman Diagrams.

It is certainly true that in the meantime "object-oriented programming" has helped to eliminate many of the problems and risks created by careless procedural programming. But due to certain other inherent problems object-oriented programs cannot always reliably guarantee the necessary response times for real-time systems (cf., e.g., Zalewski [13]). "Conventional" programs will, therefore, still be essential components of embedded systems for a long time to come.

3. Structured Real-Time Programming

3.1. General Principles

3.1.1. The Principle of the "Self-Contained Jobs"

One property of the abovementioned "structure block" appears to be of particular interest to the design of systems with parallel processes: the "principle of self-contained jobs". In classic structured programming it basically means that any significant step in the progress of a program's execution can only be started after its predecessor has been completely finished and all "side effects" been taken care of.

Applied to systems with parallel processes this principle could be extended insofar as to mean that a piece of program can do its job "without looking to the left or to the right". In particular, it does not explicitely interfere with the dynamic behaviour of other such pieces of code (e.g., by suspending or terminating them) but just emits signals that may be interpreted in such a sense by other "autonomous" pieces of code. In turn, it reacts to such signals in an appropriate way - specified by the designer. On the other hand, it has to be able to rely upon its designer to have provided all necessary resources for its normal functions and/or rules for dealing with unexpected events.

This property appears to be particularly useful in distributed systems, in systems consisting of components written in different programming languages, or in systems containing active hardware components. A (software) designer, constructing such a component, needs only to make sure that all necessary resources are available when the component becomes active and to provide rules for dealing with all possible exceptions he knows of (when designing the component), but he need not be concerned about the behaviour of other components.

Such a (self-contained) piece of code together with references to all the (static and dynamic) resources it may eventually need during its execution shall in the following be called "protoprocess".

3.1.2. The Principle of Virtual Resources

Based on research in operating systems, it can be shown that the concept of a "resource" (that is usually meant to comprise physical entities like devices or data buffers) can be generalized to include "consumeable resources" like, e.g., signals, interrupts, or events (like "buffer full"). As a consequence, they may be treated alike with respect to their reservation and use, resulting in a reduction of the number of mechanisms for the administration of resources.

A further reduction of the number of concepts - and thereby simplification - of the proposed method can be achieved by the introduction of the notion of the "virtual resource". Such a virtual resource can either be a real (permanent or consumeable) resource or one particular access right to a real resource [14]. One purpose of this approach is the possibility to separate the design of the real-time program proper from the construction of device handlers that might, e.g., allow prioritized access of processes to certain critical real resources.

3.1.3. Independence of Operating System Properties

Experience has shown that mostly the operating systems which are appropriate for real-time applications differ considerably as far as the functionality of the concrete elements for this purpose are concerned. The simplest example may be that some support semaphores, others still rely upon "signals" or "events". This may mean that an application has to be completely redesigned once it has to be ported from one operating system to another - with all the consequences of errors, retesting, etc.

Therefore, one basic principle of the proposed method is to refrain from explicitely using any of the known basic mechanisms for process interaction and manipulation, but just to express "wishes" to an operating system regarding the required behaviour of processes. The designer of the preprocessor (who usually knows better how to exploit the mechanisms of the particular OS than an application specialist) can then provide the most appropriate, reliable and efficient implementation. The application designers - who, in turn, normally are not real experts as far as details of the operating systems are concerned - can then use these mechanisms and rely on their proper functioning.

3.1.4. The Proposed Concepts

On the basis of these considerations five constructs have been identified. Three of them are related to the handling of parallel processes:

1) the **protoprocess**,
2) **structured exception handling**, and
3) the p**rocess cluster**.

The other two deal with the use of resources:

4) the **synchronization block** and
5) **integrated signalling**.

These concepts are described in detail in the following sections.

3.2. The Protoprocess

In Subsection 3.1.1 the term "protoprocess" (PPC) has been introduced, denoting a self-contained "unit of parallelism", i.e., a piece of code that may eventually (but not necessarily) be executed in parallel to other PPCs. It might also be regarded as an analogy to a "program module" (which is usually understood as a "unit of compilation"). It consists of three components:

1) the code-sequence to be executed, together with its local data;
2) the "resource claim", i.e., a list of all those virtual resources that may eventually be needed during the execution of the PPC; if a resource is used in the code sequence that has not been listed here, this is regarded as an error and, e.g., causes an error message of the preprocessor;
3) all exception handlers that may eventually be necessary to cope with the exceptions that may be raised during execution of the PPC (one of these may, e.g., be "time elapsed"); it is not allowed not to handle an exception.

The proposed graphical notation for a protoprocess (Fig. 2) is that for a code sequence (cf. Fig. 1), complemented by the resource claim.

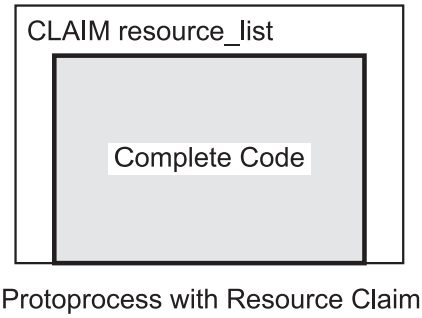

Fig. 2. Proposed graphical notation for a protoprocess.

A proposed alphanumeric syntax is:

S1 ppc-declaration ::= ppc-name **task** [**claim** formal-resource-list] [complete-code] **taskend** where "complete code" denotes the program code together with the necessary exception handlers (cf. S2, S3, S4) and "ppc-name" is the (user-defined) name by which this protoprocess can later be referenced in the program for the purpose of activating it (cf. S7). The term "formal-resource-list" has a similar meaning as the "formal parameters" in a procedure declaration, i.e., the actually used resources are only inserted at the time of activating the protoprocess - like the actual parameters during a procedure call. This also implies that the code of a protoprocess can be activated several times with a different set of virtual resources - if the underlying system allows this.

Careful readers may have observed that the text in the graphical notation in Fig. 2 (like in some following ones) does not exactly match the proposed alphanumeric syntax. It is the opinion of the authors that this is not necessary, because - like in classic structured programming - the proposed concepts (as visualized by structograms) can be implemented in slighty different ways in different programming languages. Therefore, the graphical symbols cannot represent a really binding syntax. They are rather a "graphic shorthand" for fairly complex concepts. Should, therefore, the more concrete alphanumeric syntax in this paper differ from that used in the graphic symbols, this has no technical meaning.

With regards to the proposed alphanumeric syntax a similar caveat holds: during the development and implementation of a programming language it quite often happens that the syntax of the original proposal (thoroughly as it may have been conceived) meets difficulties when implementing it for the first time and therefore will have to be modified. A really consolidated and stable syntax usually only emerges after a few test implementations have been undertaken and their results fed back into the design of the language (maybe through an appropriate committee).

3.3. Structured Exception Handling

Today, real-time programs are mostly parts of embedded systems and, therefore, during operation normally not accessible to human intervention in cases of disturbances or malfunctions. Hence, it is particularly important to provide a mechanism to cope with irregular events or "exceptions".

Fortunately, exception handling mechanisms have already been included very early in general-purpose programming languages like, e.g., in PL/I, ALGOL 68, or Ada. A particularly comprehensive and consistent proposal for exception handling in programs was published by Goodenough as early as 1975 [15] and is still valid. It has, therefore, been used as a basis for the purposes of this proposal.

As a detailed description would by far exceed the framework of this paper, Fig. 3 (left part) illustrates the basic principle: the designer of a real-time system can specify in an "exception handler" which action shall be taken in case a certain exception occurs during the execution of a process P. After completion of that action the original process can either be

- **continued** at the point of disturbance, whereby an empty exception handler means that the exception is ignored,
- **repeated** from the beginning, or
- **aborted**.

It can also

- **wait** until the cause of the exception disappears, and continue at this point. This may, e.g., be the case when a certain resource becomes available again, the temporary unavailability of which has caused the exception. Wait can also be temporarily bounded for other reasons, as shown in Subsection 3.7.1.

Of course, one might argue that this basically represents the functionality of a classic "semaphore" [16] and it would therefore be better to use this instead of the somewhat unusual mechanism proposed in this paper. However, this would be a violation of the principle described in 3.1.3: independence of operating system properties. If, e.g., an operating system does not support semaphores, the mechanism described above could be implemented in a different way and still look the same in the application program.

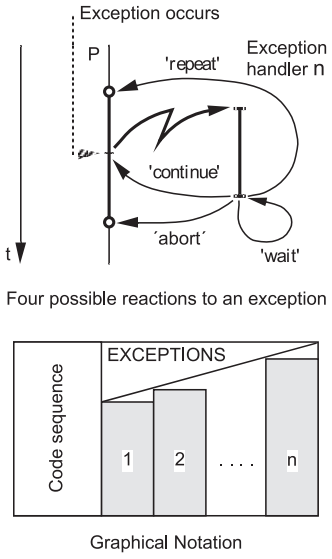

Fig. 3. Structured exception handling.

A proposed alphanumeric syntax is the following:

S2 complete-code ::= **begin** main-code [exception-handling-list **exend**] **end**.

S3 exception-handling-list ::= exception-handling-clause [{ , exception-handling-clause } . . .]

S4 exception-handling-clause ::= **on** exception-name [(exception-handler)] { **continue | repeat | abort | wait** }

The functionality of this construct can also be represented by a graphical notation (Fig. 3, right part) in a rather straightforward way. For reasons of economy of space, the figure had to be somewhat simplified. The rectangles (1 to n) shaded in grey represent the exception handlers together with their respective termination commands (continue, etc.).

3.4. The Process Cluster

Experience has shown that the proper coordination of parallel processes is an extremely demanding task for human designers and, therefore, very error-prone. Hence, it appears to be appropriate to sacrifice some flexibility in favour of predictability and simplicity.

A very early proposal has been the "fork-and-join" mechanism [17] that also appears in the form of the "parallel clause" in ALGOL 68 [18] and later in OCCAM [19]. Based on this concept the – error-prone - constructs for independent activation and termination of individual processes (like in nearly all known real-time programming languages) can be replaced by a "parallel clause" or "process cluster". By now, this is also widely known under the name of "structured parallelism".

The process cluster can be represented by means of a graphical notation, as shown in Fig. 4. The representation might be criticized for implying sequential instead of parallel execution of the protoprocesses, but its design had to be a compromise between rigour, readability, and the necessity to fit into nested structure diagrams.

A proposed alphanumeric syntax for this mechanism is the following:

S5 parallel-clause ::= **parallel** ppc-activation-list **parend**

S6 ppc-activation-list ::= ppc-activation [{, ppc-activation} . . .]

S7 ppc-activation ::= **execute** ppc-name [**with** actual-resource-list] [**priority** priority-specification]

where **ppc-name** denotes the name of a protoprocess, **actual-resource-list** is the list of resources the ppc is actually using during its execution as a process, and **priority-specification** a value for the priority of that process after its activation.

In the form described in this paper, the parallel statement block can only include execute statements. Of course, one could imagine that it might as well directly include ppc-declarations. In such a case, these would not even need a ppc-name any more. However, such a solution would require that the entire construct be written in the same programming language. One would lose the possibility of putting together a system from components written in different programming languages or even implemented as separate hardware (as e.g. mentioned in 3.1.1).

3.5. The Synchronization Block

It turned out that all cases of access to resources can be expressed by a construct that operates like a "critical section" with the modification that a set of exception handlers (cf. 3.3) specifies what is to happen if a resource is unavailable, breaks down, is requested by another process with higher priority, etc. According to the considerations in Section 3.1.2 this holds for all types of virtual resources.

By means of proper rules it can also be made sure that resources are released in an appropriate way after their use. Compliance with these rules can even be checked before runtime. Thereby, many very frequent program-

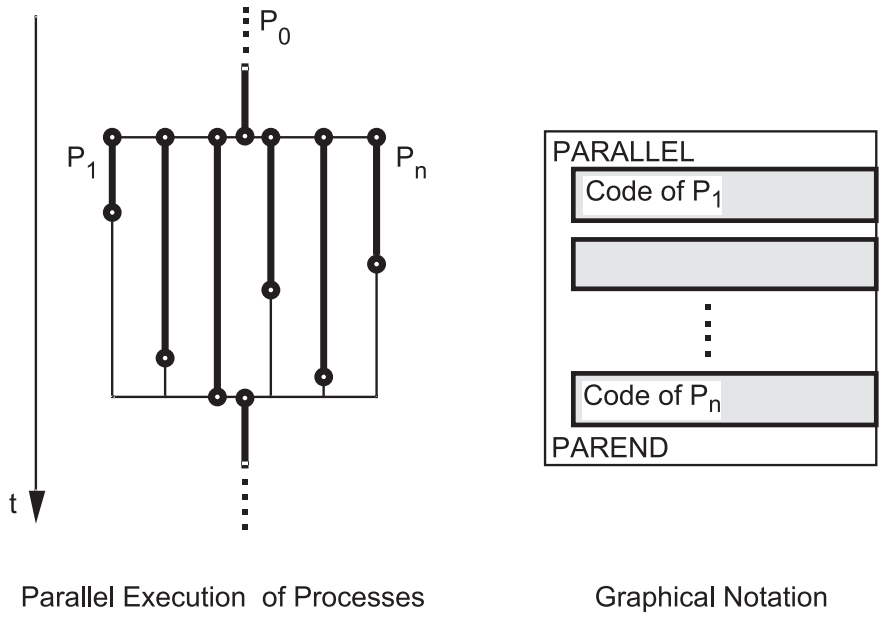

Fig. 4. Strictly controlled parallelism.

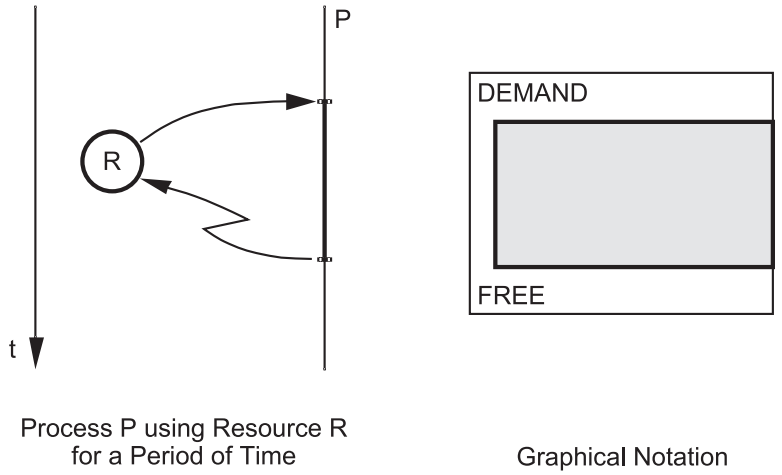

Fig. 5. Use of resources.

ming errors and runtime problems can be avoided like, e.g., deadlocks that result from blocking and releasing resources in the wrong order. It should be mentioned that all resources are released if the demand block cannot be executed because one resource is not available.

Fig. 5 illustrates this behaviour and shows the respective graphical notation. As far as the choice of the keyword "demand" is concerned, it should be noted that it is a consequence of the underlying concept of "virtual resources" (cf. 3.1.2). In this case it means that an individual programmer (e.g., in a design team) cannot reserve a resource for exclusive use without any precautions – and, thereby, eventually interfere with the system's resource administration mechanism, or with a safety-oriented allocation strategy that has been agreed upon in advance. He has to make sure that he has the (access) right to do so.

A proposed alphanumeric syntax is the following:

S8 resource-reservation ::= **demand** resource-list [main-code] **free**

3.6. Integrated Signalling

A frequent class of runtime problems in multi-process systems is caused by the fact that a process waits for the completion of an action of another process which, in turn, is interrupted by still another process just before it can signal the completion of that action. Even in the case that the processes communicate correctly by means of semaphores, there may be unnecessary delays for the waiting process.

One reason for this problem is that - despite of the fact that correctly implemented semaphore operations proper are uninterruptible - the execution of the program, containing these operations, can be interrupted just before the statement containing the semaphore operation is reached. It, therefore, appears appropriate to introduce a kind of "higher-level uninterruptiblity" of the code sequence containing the release of a signal.

These considerations led to the definition of another construct, the "integrated signalling" (Fig. 6). It is specified as follows: if a certain code sequence has been successfully completed, a signal is emitted ("a consumable resource is created"). This can be consumed like any other resource by other processes using the demand statement. The behaviour of this construct is comparable to the uninterruptibility of the classical semaphore operation - but on a higher level. Emit enters a code section only if the consumable resource does not currently exist. The resource is created when "emend" is executed.

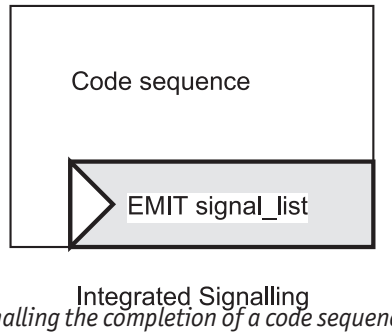

Fig. 6. Signalling the completion of a code sequence.

The proposed alphanumeric syntax is:

S9 signal-emission ::= **emit** signal-list [main-code] **emend**

3.7. Potential of the Proposed Constructs

3.7.1. Treatment of Deadlines and Timeouts

One of the most important features of real-time programs is their ability to cope with the fact that the time allowed for the execution of a process is running out, or that something has to be done if a process has waited too long for the availability of a resource. The following two examples shall illustrate how this can be achieved with the proposed constructs.

In both cases it is assumed that the underlying operating system contains a mechanism, e.g., called "timer", that can be set by the program and emits a signal after this preset time interval has elapsed.

The first example (Fig. 7) illustrates the case that the execution time of a process for some reason exceeds the allowed limit.

The second example (Fig. 8) shows how the case can be handled when a resource does not become available after a certain time and some alternative action has to be taken.

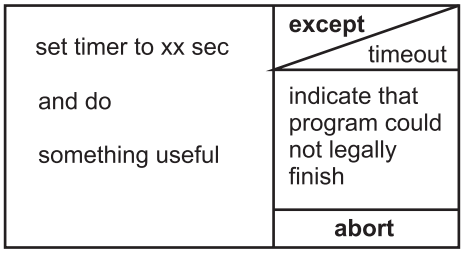

Fig. 7. Supervision of process timeout.

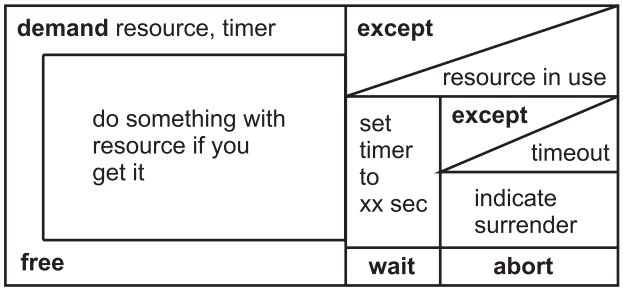

Fig. 8. Quit waiting for a resource after a predefined time interval.

3.7.2 Reaction to Emergency Conditions

A thorough analysis of the potential of such a complete exception handling mechanism shows that it can compensate the presumed rigidity of the parallel clause. A combination of both constructs results, therefore, in a mechanism - the "emergency group" - capable of describing practically all possible dynamic structures in real-time systems in a very flexible way. Nevertheless, it leads to a predictable behaviour of the program system.

The underlying principle is that the processes of a computing system controlling a technical system can basically be divided into three classes:

Class 1 processes which always have to run - whatever happens to the technical system;

Class 2 processes which have to be terminated in case of an emergency (e.g., breakdown of a vital component of the technical process under supervision), because they are either useless or even constitute a risk and, therefore, all have to listen to the same **globally triggered** exception (e.g., temperature too high) and react by terminating them-selves, after which their parallel clause is terminated as well and the processes of class 3 start;

Class 3 processes which are designed to deal with this emergency and, therefore, have to be activated in that case.

These classes then form three separate process clusters that are grouped according to the principle shown in Fig. 9:

Cluster a contains the processes of class 2, equipped with exception handlers that first deal with the exception raised in that particular case of emergency and then finish with "abort";

Cluster b contains the processes of class 3, which are activated in that case and terminate after the emergency has been dealt with.

Clusters a and b are then surrounded by a "repeat block" that makes sure that the processes of class 2 are reactivated after the disappearance of the emergency.

Cluster c finally contains the processes of class 1 together with this repeat block.

In Fig. 10 it is tried to give an impression how this construction can be realized using the proposed concepts. For reasons of limited space it has been restricted to process cluster c, the repeat clause, containing clusters a and b, and one class 2 process.

This structure makes sure that reactions of the program system to emergencies in a technical system under supervision and control are planned in advance in a clear and consistent manner. Thus, an important criterion for the design of robust program systems is fulfilled.

At runtime, the structure guarantees that no unplanned or uncoordinated (re)actions onto the technical process can be initiated by computing processes that were left in an undefined state at the time of occurrence of the emergency. This fulfils another important criterion for predictable and reliable system behaviour.

Of course, in practical applications there may be more

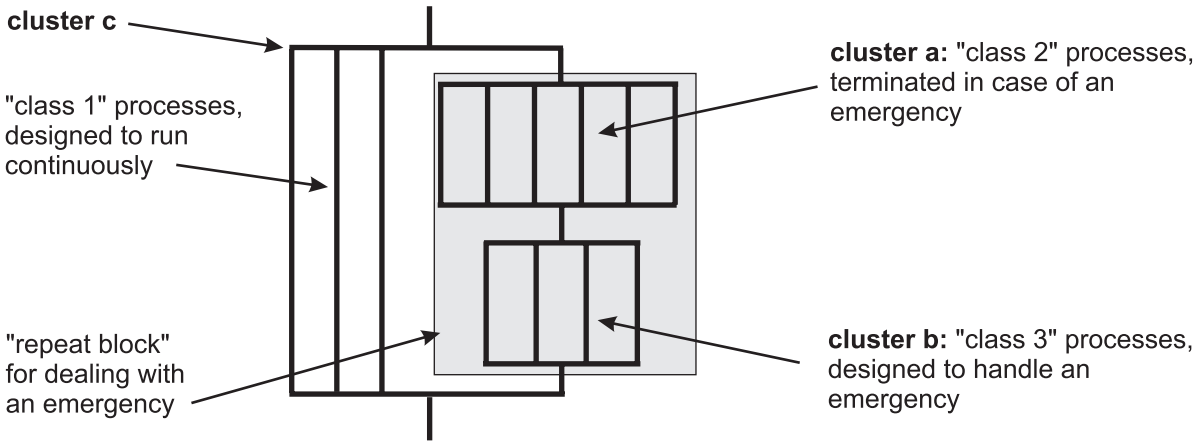

Fig. 9. The Emergency Group.

than one "fatal exception" in the system and therefore the pre-planned reactions will have to be more complex than in this example. However, the authors hold that the rigidity of the proposed method will be beneficial in terms of reliability of the resulting system because it will force designers to plan the dynamic structures much more in advance than it is usually done today. In particular, there will be much less ad-hoc reactions that are patched into the code "in hindsight" and which are therefore extremely error-prone. Thus, they trust, the proposed method will considerably contribute to the reliability of the resulting system.

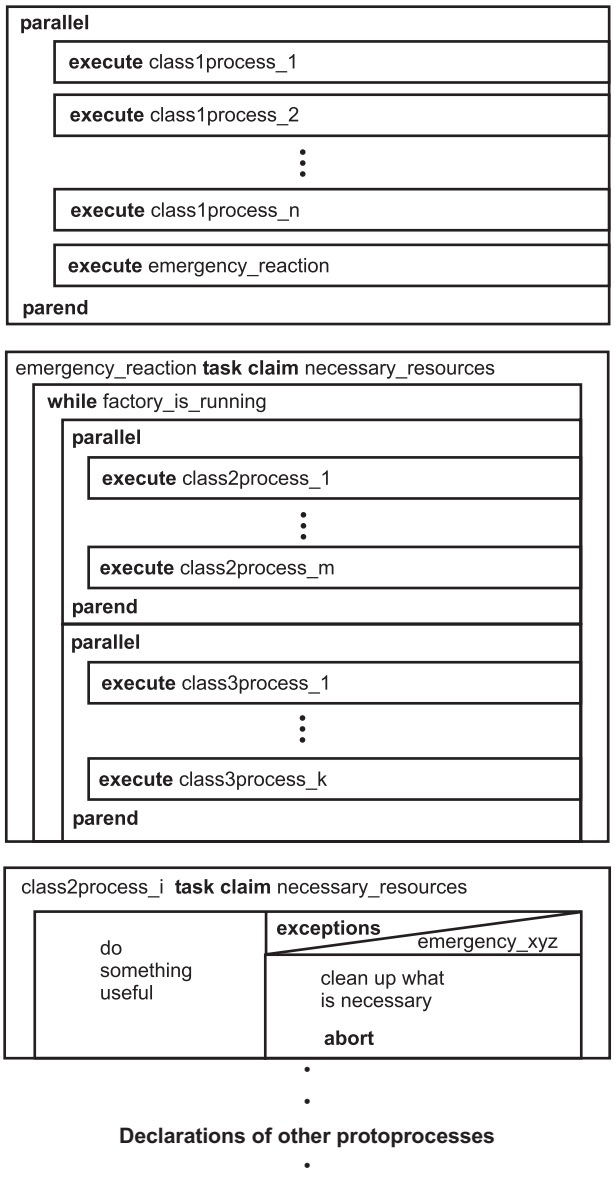

Fig. 10. The Principle of the Emergency Group in Graphical Syntax.

3.7.3. Deadlock Prevention

Finally, it is necessary to make sure that a real-time system for critical applications is free of deadlocks. It is known that deadlock detection and prevention mechanisms can be applied to a program system if each process indicates its intended use of resources prior to its execution. The resource claim, introduced in Section 3.2, can be used for this purpose (cf. Fig. 2). On this basis a deadlock prevention technique can be integrated in the proposed method. It is based on a proposal by Habermann [20], that can be summarized as follows:

- arrange all resources that will be needed during the execution of a program system in a linear order,
- use them only in this order, and
- release them in reverse order.

In order to achieve this, the declaration of a protoprocess has to contain a list of all resources that will eventually be needed during its execution in the form of a resource-claim. In a demand-clause, then, of course only those resources, that are actually needed, are listed in the same order. By means of this rule, already a preprocessor can check whether this correct sequence is followed, and can eventually even enforce that the resources are released in the correct order. As a result, the application of the design method presented in this paper practically guarantees that the resulting real-time systems are free of deadlocks. This is a very important aspect for real-time systems that have to work autonomously.

Of course, the authors are aware of the fact that this is a very strict rule, which cannot always be followed easily. But sometimes the only remaining alternative might be an unreliable system behaviour - and that should of course be avoided. However, if a system designer bears that rule in mind already during early states of system development and tries to avoid such situations, it can lead to a better overall architecture of the system.

4. A Test Implementation

4.1. Technical Basis

As a feasibility test, the proposed constructs have been implemented in the framework of a student project [1], comprising two persons and approximately three person months. Although this narrow time-frame and the shortage of financial means required some technical compromises regarding the completeness of the implementation, the results can be regarded as a success.

The ideal implementation of the proposed method would have been an interactive programming tool on a graphical basis. But this would have required much too high an effort. Therefore, a text-oriented method on the basis of a macropreprocessor was chosen. Because the preprocessor available for "C" was not powerful enough, "M4"© from the UNIX© Environment was employed.

The following target languages were selected:

- "C", because it is currently the most widely used language for system programming, and
- PEARL, because this language has explicitly been developed for real-time programming.

The implementation for "C" was based on "Realtime-C M7©" on a Siemens process control computer with the operating system "Rmos32©". The implementation for PEARL used a Windows©-PC with the operating system "RTOS-UH©" of the University of Hanover.

A programmer who wishes to use the macros would create his programs with any editor or IDE he chooses,

feed his finished program files (which use the macros) to "M4"© and, then, feed the "M4"© output to the C or PEARL compiler. PEARL programs need to include the RTS.ph file, Realtime-C programs include the RTS.h file.

Two different sets of macros were written, one for each of the chosen programming languages. Both have, however, the same syntax and the same functionality. The challenge was to find a common base of functionality between C (allows the user to do almost anything) and PEARL (focusses on readability, but does not allow the same things as other languages). It could be shown that most of the required functionality is available (more or less directly) in both languages.

Implementations for "Java©" und "Ada©" were also taken into consideration, but could not be realized because of lack of time. It was also discussed whether a suitable object-oriented language could be used as a basis. However, some doubts concerning the predictability of the time behaviour of object-oriented programs remained (cf. Section 2.3).

As examples for necessary modifications two cases shall shortly be mentioned:

- The proposed notation for exception handling could not quite be realized. The exception macro cannot refer to arbitrary preceeding blocks, because e.g. labels and counters for the signal-group-reaction have to be initialized. Therefore, similar to usual notations (C++, Java), the macro "Try" was introduced, that opens a block containing exceptions. Additionally, the syntax for exceptions had to be modified.
- Another example for an indirect implementation of a macro is the parallel tasks macro: the version of Realtime-C that has been used for the test implementation only supports binary semaphores. However, if two or more tasks end before ParEnd requests the semaphore of the first task (cf. Subsection III.1), it will be released more than once. But, as its value cannot be increased to more than one, tasks have to request a second semaphore to be allowed to finish themselves.

The implementation was tested and demonstrated by means of two examples:

- a producer-consumer problem (4.2.1) and
- sending and processing of signals (4.2.2).

These are described in the following section.

As far as the syntax of some of the implemented constructs is concerned, the reader will note that it differs in some places from the proposed form, because not all concepts could syntactically be implemented as originally proposed. The authors, however, chose to leave these inconsistencies visible instead of "covering them up" by modifiying the proposed syntax in order to fit this first experimental implementation. They would rather encourage a process of several test implementations in order to find out the best way of implementing the proposed concepts and arriving at a really consolidated syntax by a proper process of discussion among interested people.

In order to encourage interested software designers to test the implemented macros, they have been made publicly available on the Internet [21]. They can be used under the terms and conditions of the GNU General Public Licence [22].

4.2. Demonstration Examples

4.2.1. The Producer-Consumer Problem

Basically, this problem consists of two processes: a producer process that calculates the squares of natural numbers, and a consumer process that displays the results on a screen. The results to be displayed are passed from the first process to the second one using a (non-protected) buffer in memory. For the synchronization between the two processes signals are used, because no critical behaviour of this miniature system is to be expected.

The following parts of the entire program shall now give an impression as to how programs look alike in the proposed notation and, how much their lengths increase after expansion of the macros. In particular, it can be seen how much more lines of code would have to be written in a conventional programming language in order to achieve a comparable functionality which, of course, would result in a proportionally higher number of programming errors (cf. Section 2.2).

Using the macros developed in the student project, the code of this program example is the following:

I) Code in PEARL

```
.
Exclusive (Display, Disp);     // exclusive resource
Signal (Sig);                  // signal
.
```

I. 1) The Producer Process

```
Task(Producer) Claim(Sig);
   FOR i FROM 0 TO MaxValue REPEAT
      Try;
         Emit(Sig);
            Value = i * i;
         EmEnd;
      Exceptions(EXCEPTION_SignalInUse);
         Wait;
      ExEnd;
   END;
TaskEnd;
```

I.2) The Consumer Process

```
Task(Consumer) Claim(Sig, Display);    DCL ValueCopy FIXED INIT(0);
   WHILE ValueCopy < MaxValue*MaxValue REPEAT
      Try;
         Demand(Sig, Display);
            ValueCopy = Value;
            PUT Value, NL TO Disp;
         Free;
      Exceptions(EXCEPTION_ResourceInUse, EXCEPTION_SignalNotSent);
         Wait;
      ExEnd;
   END;
TaskEnd;
```

II) Code in C

```
.
Exclusive (Display, stdout);    // exclusive resource
Signal (Sig);                   // signal
.
```

II.1) The Producer Process

```
TASK(Producer) Claim(Sig) {
   int i ;
   for (i = 0; i <= MaxValue; ++i) {
      Try;
         Emit(Sig);
            Value = i * i;
         EmEnd;
      Exceptions(EXCEPTION_SignalInUse);
         Wait;
      ExEnd;
   }
} TaskEnd;
```

II.2) The Consumer Process

```
Task(Consumer) Claim(Sig, Display) {
   int ValueCopy = 0;
   while (ValueCopy < MaxValue * MaxValue) {
      Try;
         Demand(Sig, Display);
            ValueCopy = Value;
            fprintf(stdout, "%d\n", Value);
         Free;
      Exceptions(EXCEPTION_ResourceInUse, EXCEPTION_SignalNotSent);
         Wait;
      ExEnd;
   }
} TaskEnd;
```

As can be seen from these examples, the written code is very short and easy to read, although it contains rather powerful constructs for process coordination and resource administration.

In order to give an impression of the amount of code programmers would have to write in order to achieve the same quality of the coordination mechanisms, parts of the code, that has been produced by means of the macrogenerator, are shown in the next section.

III) Expanded Code in PEARL

III.1) The Producer Process

```
Producer:TASK;
   FOR i FROM 0 TO MaxValue REPEAT
      BEGIN;
         DCL (RTS_ContinueAt1,RTS_ThrownException1,RTS_Wait1) FIXED INIT(0,0,0);
         RTS_TryLabel1:
         CASE RTS_ThrownException1 ALT(0);
            IF RTS_Wait1 == 0 THEN
               IF NOT(TRY(EmitSema_Sig)) THEN
                  RTS_ContinueAt1 = 1;
                  RTS_ThrownException1 = 1;
                  GOTO RTS_TryLabel1;
                  RTS_ThrowLabel1_1:;
               FIN;
            ELSE
               REQUEST EmitSema_Sig;
            FIN;
            Value = i * i;
            RELEASE ResourceSema_Sig;
         ALT(1);
            RTS_Wait1 = 1;
            RTS_ContinueAt1 = 0;
            RTS_ThrownException1 = 0;
            GOTO RTS_trylabel1;
         FIN;
         RTS_ExEndLabel_1:
      END;
   END;
   RELEASE RTS_TaskSema(1);
END;
```

Remark: the Semaphore "RTS_TaskSema" is necessary in the parend-clause of the main function. Its purpose is to detect that all parallel tasks have been completed. It is, therefore, automatically generated and inserted by the macro generator.

III.2) The Consumer Process

```
Consumer:TASK;
   DCL ValueCopy FIXED INIT(0);
   WHILE ValueCopy < MaxValue * MaxValue REPEAT
      BEGIN;
         DCL (RTS_ContinueAt2,RTS_ThrownException2,RTS_Wait2) FIXED INIT(0,0,0);
         RTS_TryLabel2:
         CASE RTS_ThrownException2 ALT(0);
            IF RTS_Wait2 == 0 THEN
               IF NOT(TRY(ResourceSema_Sig)) THEN
                  RTS_ContinueAt2 = 1;
                  RTS_ThrownException2 = 3;
                  GOTO RTS_TryLabel2;
                  RTS_ThrowLabel2_1:;
               FIN;
            ELSE
               REQUEST ResourceSema_Sig;
            FIN;
            IF RTS_Wait2 == 0 THEN
               IF NOT(TRY(ResourceSema_Display)) THEN
                  RELEASE ResourceSema_Sig;
                  RTS_ContinueAt2 = 2;
                  RTS_ThrownException2 = 2;
                  GOTO RTS_TryLabel2;
                  RTS_ThrowLabel2_2:;
               FIN;
            ELSE
               REQUEST ResourceSema_Display;
```

```
          FIN;
          ValueCopy = Value;
          PUT Value, NL TO Disp;
          RELEASE ResourceSema_Display;
          RELEASE EmitSema_Sig;
        ALT(2,3);
          RTS_Wait2 = 1;
          RTS_ContinueAt2 = 0;
          RTS_ThrownException2 = 0;
          GOTO RTS_TryLabel2;
        FIN;
        RTS_ExEndLabel_2:
      END;
   END;
   RELEASE RTS_TaskSema(1);
END;
```

IV) Expansion of the Producer Process in C

```
void _FAR _FIXED Producer(void) {
   do {
      int i;
      for (i = 0; i < maxvalue; ++i) {
         do {
            int RTS_ContinueAt = 0, RTS_ThrownException = 0, RTS_Wait =0;
            RTS_TryLabel1:
            switch (RTS_ThrownException) {
               case(0):
                  do {
                     if (RTS_Wait1 == 0) {
                        if (RmGetbinSemaphore (RM_CONTINUE, EmitSema_Sig)!=RM_OK) {
                           RTS_Continueat = 1;
                           RTS_ThrownException = 1;
                           goto RTS_TryLabel1;
                           RTS_ThrowLabel1_1:;
                        }
                     } else {
                        RmGetBinSemaphore(RM_WAIT, EmitSema_Sig);
                     }
                  } while (0);
                  Value = i * i;
                  RmReleaseBinSemaphore(ResourceSema_Sig);
                  break;
               case 1:;
                  /* In case the signal is used,*/
                  /* wait until it is free again*/
                  RTS_Wait = 1;
                  RTS_ContinueAt = 0;
                  RTS_ThrownException = 0;
                  goto RTS_TryLabel1;
                  break;
            }
         }while(0);
      }
   } while(0);
   RmGetBinSemaphore(RM_WAIT, RTS_TaskReleaseSema[0]);
   RmReleaseBinSemaphore(RTS_TaskSema[0]);
   RmDeleteTask(RM_OWN_TASK);

};
```

The consumer process in C has been left out in order to save space. It consists of still more lines of code than the expanded example in PEARL, and the authors believe that the character of the proposed method is sufficiently illustrated by the examples given.

4.2.2. The Signal Transmission Problem

This example demonstrates the transmission of signals from one computer to another one. On the first computer coordinates are input by means of a mouse, on the second one they are displayed graphically. It was chosen, because the user interaction and visualisation of the input reflect some real-time requirements. In this case, they are the result of the necessary interaction with a human user. Therefore, unbounded wait statements are adequate. However, these should not be applied in hard real-time applications, where continue, repeat and cancel offer better predictability of the program execution behaviour.

Fig. 11 shows the structograms used for the specification of the program code in order to give an impression of the applicability of the proposed graphical notation. Unfortunately, the example could only be programmed in PEARL, because the process control computer (which could run "C") did not have the necessary equipment.

4.3. Discussion of Results

As already mentioned in Section 4.1, the effort for the test implementation of the preprocessor only amounted to approximately three person months. In this short time more of the proposed concepts were implemented than had been expected. To the opinion of the authors this compares rather favourably with the usual implementation costs of software tools.

As could be expected, some problems arose with respect to a complete implementation of the proposed constructs in a one-to-one fashion. But they were also smaller than expected. Most of the necessary modifications were related to the syntax, because a macropreprocessor is less powerful in that respect than an interactive graphical programming tool.

However, the expected reduction of the complexity of the programming process has been achieved. First of all, the length of the programs to be written is typically more than 50% shorter than until now, when the described programming mechanisms would have to be programmed manually - and provided the same reliability-oriented mechanisms. Of course, one might argue that part of this reduction could have been achieved without the proposed constructs by means of the use of macros alone. However, the authors hold that the much more flexible character of "classic" programming elements for real-time applications (according to experience) usually leads to many different ("individualistic") implementations of the same functionality and therefore would not really allow a comparable implementation in the form of macros. Therefore, according to the well-known direct relationship between the number of written "lines of source code" and the number of errors in a program this already means a considerable increase in the reliability of the resulting programs. In addition, the proposed constructs are much easier to comprehend than conventional real-time programs. This leads to a further reduction of design errors.

In addition to the benefits that have been mentioned in Section 2.2, discussion turned out that some further means (cf. Section 2.1) for the support of reliability are provided by the proposed method:

It is possible to model the behaviour of a software system in order to discover conceptual errors. For this purpose one could, e.g., imagine a preprocessor that separates the real-time constructs from the algorithmic parts and replaces these by dummies, containing runtime estimates. The result could then be used as input for a simulator. This possibility of modelling the real-time behaviour of a system also allows - to a certain degree - fault forecasting, at least as far as timing and synchronization problems are concerned.

Last but not least, the method supports (as far as can reasonably be expected) software re-use which, in turn, helps in fault prevention. The basis for this is the separation of the computational parts of a program from the ones determining its real-time behaviour. On this basis, it is either possible to re-use tested and proven algorithms in a new real-time framework, or to replace outdated algorithms in a stable real-time structure without running the risk of disturbing or destroying critical timing relations.

5. Conclusions

By means of the test implementation presented in this paper, it could be demonstrated that the proposed constructs indeed work, and can be applied to the construction of reliable real-time programs.

It could be shown that the proposed "real-time structograms" can be applied to a variety of programming languages, and that they allow the specification of real-time systems independent of the underlying operating system. Therefore, the authors hold that this method is an excellent means for the design of safe and reliable real-time systems in the future, because it does not affect the customary use of common programming languages, operating systems, or hardware product lines.

A very interesting feature appears to be the possibility to extract information from the design of a real-time system that can be used as input for simulators or queuing models in order to test the behaviour of a real-time system under design already in very early phases.

A closer look even reveals that it can also be used for the design of entire systems - including hard-ware components. The reason for this is that the abstraction level of the structural elements is so high that they can also represent the functions of hardware components. Basically, this is already a property of structograms in general. These do not describe a flow of control as a sequence of detailed single statements, but more or less in the form of "action groups". Such an action group could, therefore, be implemented as a completely independent program, running on some remote processing unit. This, in turn, may well be implemented in the form of a custom-made chip.

The process cluster is a still more instructive example: the individual processes of a cluster might be running on completely separate processing units. It is only necessary

that a mechanism exists for starting them at the same time, and for making sure that their termination is coordinated according to the rules of the flow of control "of higher order", of which they are part according to the specification given by the overall structogram. However, a complete description of this approach would require a separate paper.

The authors hope that the presented method turns out to be a useful contribution to the solution of some problems connected with the design of real-time systems and look forward to comments and discussions. Finally, they want to thank the reviewers for their numerous detailed and useful comments. These certainly helped to identify a number of points that had not been made completely clear in the first place. The authors hope that this revised version now gives the reader a sufficiently complete impression.

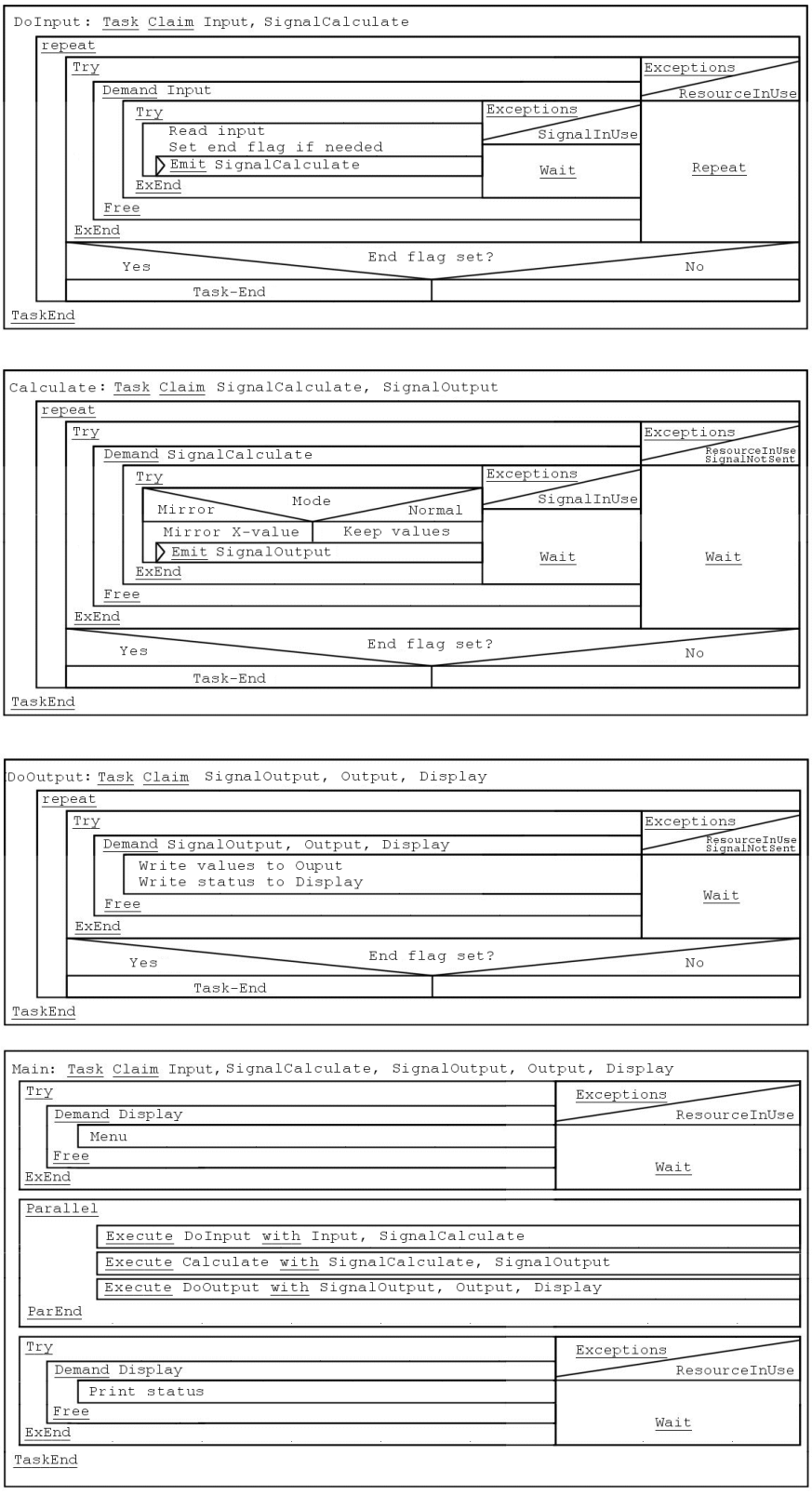

Fig. 11. Structograms of the Signal Transmission Problem.

AUTHORS
Peter F. Elzer* - retired director of the Institute for Process and Production Control Technology, Technical University of Clausthal, now living at: Paradiesstr. 4, D-80538 München, Germany. E-mail: elzer.home@t-online.de.
Martin Gollub - Werum Software & Systems AG, Wulf-Werum-Straße 3, D-21337 Lüneburg, Germany. E-mail: martin.gollub@werum.de.
Sven Trenkel - Schlesienstr. 7, D-21391 Reppenstedt, Germany.
* Corresponding author

References

[1] Trenkel S., Gollub M., *Implementation von Methoden zur zuverlässigen Programmierung von Echtzeitsystemen (Implementation of Methods for Reliable Programming of Real-Time Systems)*; Student Project in the Department of Mathematics and Computer Science of the Technical University of Clausthal Prof. K. Ecker, 2004.

[2] Elzer P., "A Mechanism for the Design of Structured Real-time Programs for Process Automation"; Conference Proceedings "Prozessrechner 77", *Informatik-Fachberichte*, vol. 7, Springer-Verlag: Berlin-Heidelberg-New York, 1977, pp. 137-148.

[3] Elzer P., *Structured Description of Process Systems*; Dissertation, Report of the Institute for Mathematical Machines and Data Processing of the University of Erlangen-Nuernberg, vol. 12, no. 1, 1979.

[4] Elzer P., "Missed Opportunities in Real-time Programming?". In: Wolfinger B. (ed.), *Innovationen bei Rechen- und Kommunikationssystemen, Proceedings of the 24th Annual Conference of the German Computer Society in the framework of the 13th IFIP World Congress*, Springer-Verlag: Berlin-Heidelberg-New York, 1994, pp. 328-339.

[5] Elzer P., "A Method for the Construction of Reliable Real-Time Programs". In: *Proceedings of the Conference on Embedded Systems*, Munich, 2004.

[6] DIN 66253-2 PEARL 90; Beuth Verlag, Berlin, Köln, 1998.

[7] Frevert L., Beschreibung der PEARL90-Syntax (Description of the Syntax of PEARL); *http://www.irt.uni-hannover.de/pub/pearl/Frevert/PEARL90Syntax.pdf*

[8] Siemens SIMATIC Software, System Software for M7-300/400, System and Standard Functions; Reference Manual, 1997.

[9] Laprie J.-C., Dependability of Computer Systems: from Concepts to Limits; 1998 IFIP International Workshop on Dependable Computing, Johannesburg.

[10] Industrial-process measurement and control - Evaluation of system properties for the purpose of system assessment. Part 5: Assessment of system dependability, Draft, Publication 1069-5, CEI Secretariat, 1992.

[11] Parnas D.L., A Technique for Software Module Specification with Examples; *Comm. ACM*, vol. 15, no. 5, 1972, pp. 330-336.

[12] Nassi B., Shneiderman A., "Flowchart Techniques for Structured Programming", *SIGPLAN Notices*, vol. 8, no. 8, 1973, pp. 12-26.

[13] Zalewski J., "Object-Orientation vs. Real-Time Systems", *Real-time Systems Journal*, vol. 18, 2000, pp. 75-77.

[14] Elzer P., "Resource Allocation by Means of Access Rights, an Alternative View on Real-time Programming". In: *Proceedings of the IFAC/IFIP Workshop on Real-time Programming*, Pergamon Press: Oxford-New York, 1980, pp. 73-77.

[15] Goodenough J.B., "Exception Handling - Issues and a Proposed Notation", *Comm. ACM*, vol. 18, no. 12, 1975, pp. 683-696.

[16] Dijkstra E.W., "Cooperating Sequential Processes". In: Genuys (ed.) *Programming Languages*, London, 1969.

[17] Van Horn D., "Programming Semantics for Multiprogrammed Computations", *Comm. ACM*, vol. 9, no. 3, 1966, pp. 143-155.

[18] Van Wijngarden A., Mailloux B.J., Peck J.E.L., Koster C.H.A. "Report on the Algorithmic Language AL-GOL 68", *Num. Math.*, vol. 14, 1969, pp. 79-218.

[19] Jones G., *Programming in OCCAM*, Prentice Hall International: Englewood Cliffs, London, 1987.

[20] Habermann N., Prevention of System Deadlocks; *Comm. ACM*, vol. 12, no. 7, 1969, pp. 373-385.

[21] *http://home.tu-clausthal.de/student/software/*

[22] *http://www.gnu.org/licenses/gpl.txt*