Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

On Issues of Constructing an Exception Handling Mechanism for CSP-Based Process-Oriented Concurrent Software†

Dusko S. JOVANOVIC, Bojan E. ORLIC, Jan F. BROENINK
Twente Embedded Systems Initiative,
Drebbel Institute for Mechatronics and Control Engineering,
Faculty of EE-Math-CS, University of Twente,
P.O.Box 217, 7500 AE, Enschede, the Netherlands
d.s.jovanovic@utwente.nl

Abstract. This paper discusses issues, possibilities and existing approaches for fitting an exception handling mechanism (EHM) in CSP-based process-oriented software architectures. After giving a survey on properties desired for a concurrent EHM, specific problems and a few principal ideas for including exception handling facilities in CSP-designs are discussed. As one of the CSP-based frameworks for concurrent software, we extend CT (Communicating Threads) library with the exception handling facilities. The extensions result in two different EHM models whose compliance with the most important demands of concurrent EHMs (handling simultaneous exceptions, the mechanism formalization and efficient implementation) are observed.

Introduction

Under *process-oriented* architectures in principle we assume that a program's algorithms are confined within *processes* that exchange data via *channels*. When based on CSP [1], channels (communication relationships) are synchronous, following the rendezvous principle; executional compositions among processes are ruled by the CSP constructs, possibly represented as compositional relationships [2]. Today's successors of the programming language occam, which was first to implement this programming model, are occam-like libraries for Java, C and C++ (the most known are the University of Twente variants CTJ [3], CTC and CTC++ [2, 4] and the University of Kent variants JCSP [5], CCSP [6] and C++CSP [7]). "Twente" variants are together referred to as CT (Communicating Threads), and for this paper all experiments are worked out within that framework. The general Twente CSP-based framework for concurrent embedded control software is referred to as CSP/CT, which implies the use of those concepts of CSP that are implemented in the CT and accompanying tools [8] in order to provide this particular process-oriented software environment.

Recent work [9] is concerned with dependability aspects of the CSP/CT, which revives interest in fault tolerance mechanisms for CSP/CT, and among them the exception handling mechanism (EHM). Exception handling is considered "as the most powerful software fault-tolerance mechanism" [10]. An exception is an indication that something out of the ordinary has occurred which must be brought to the attention of the program which raised it [11]. Practical results during the research history of thirty years ([12]) appeared as

†This research is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

sophisticated EHMs in modern mainstream languages used for programming mission-critical systems, like C++, Java and Ada. This paper considers the exception handling concept on a *methodological level* of designing concurrent, CSP/CT process-oriented software.

An EHM allows system designers to distribute dedicated corrective or alternative code components at places within software composition that maximize effectiveness of error recovery. Principles of EHM are based on provision of separate code segments or components to which the execution flow is transferred upon an error occurrence in the ordinary execution. Code segments or components that attempt error recovery (*exception handling*) are called *exception handlers*. The main virtue of this way of handling errors in software execution is a clear separation between normal (*ordinary*) program flow and parts of software dedicated to correcting errors.

Because of alterations of a program's execution flow due to exceptional operations, EHMs additionally complicate understanding of concurrent software. In [13] issues of exception handling in sequential systems are contrasted with those in concurrent systems, especially the problems of *concurrently raised exceptions* resolution and *simultaneous error recovery*.

Despite favourable properties in structuring error handling and the fact that EHM is the only structured fault tolerance concept directly supported at the level of languages, it is not so readily used in mission- or life-critical systems. Lack of tractable methods for testing or, even more desired, formal verification of programs with exception handling is to be blamed for hesitant use of this powerful concept. As clearly stated in [14], "since exceptions are expected to occur rarely, the exception handling code of a system is in general the least documented, tested, and understood part. Most of the design faults existing in a system seem to be located in the code that handles exceptional situations."

1 Properties of Exceptions and Exception Handling Mechanisms (EHMs)

1.1 EHM Requirements

1.1.1 General EHM Properties

The following list combines some general properties for evaluating quality and completeness of an *Exception Handling Mechanism* (EHM) [13, 15, 16]. It should:

1. be simple to understand and use.
2. provide a clear separation of the ordinary program code flow and the code intended for handling possible exceptions.
3. prevent an incomplete operation from continuing.
4. allow exceptions to contain all information about error occurrence that may be useful for a proper handling, i.e. recovery action.
5. allow overhead in execution of exception handling code only in the presence of an exception – exception handling burdens on the error-free execution flow should be neglectable.
6. allow a uniform treatment of exceptions raised both by the environment and by the program.
7. be flexible to allow adding, changing and refining exceptions.
8. impose declaring exceptions that a component may raise.
9. allow nesting exception handling facilities.

1.1.2 Properties of a Concurrent EHM

The main difficulty of extending well-understood sequential EHMs for use in concurrent systems is the effect that occurrence of an exception in one of the collaborating processes certainly has consequence to the other (parallel composed) processes. For instance, exceptional interruption in one process before a rendezvous communication certainly causes blocking of the other party in the communication, causing a deadlock-like situation [17]. It is likely that an exceptional occurrence detected in one process is of concern of the other processes.

In large parallel systems it may easily happen that independent exceptions occur simultaneously: more than one exception had been raised before the first one has been handled. The EHM, actually exception handlers, should detect these so-called *concurrent exception occurrences* [13]. Also the same error may affect different processes during different scenarios, so causing different but related exceptions. Such concurrent (and possibly related) exceptions need to be treated in a holistic way. In these situations handling exceptions one-by-one may be wrong – therefore in [13] the notion of *exception hierarchy* has been introduced. The term "exception hierarchy" should be distinguished from the *hierarchy of exception handlers* (which determines exception propagation, as addressed in the remainder). Neither has it anything to do with a possible inheritance hierarchy of exception types. The concept of exception hierarchy helps reasoning and acting in the case of multiple simultaneously occurring exceptions: "if several exceptions are concurrently raised, the exception used to activate the fault tolerance measures is the exception that is the root of the smallest subtree containing all of the exceptions" [13].

For coping with the mentioned problems, a concurrent EHM should make sure that:

10. upon an exception occurrence in a process communicating in a parallel execution with other processes, all processes dependent on that process should get informed that the exception has occurred.
11. all participating processes simultaneously enter recovery activities specific for the exception occurred.
12. in case of concurrent exception occurrences in different parallel composed processes, a handler is chosen that treats the compound exceptional situation rather than isolated exceptions.

1.1.3 Formal Verifiability and Real-time Requirements

In order to use any variant of the EHM models proposed in section 3, for high integrity real-time systems (and to benefit from the CSP foundation for such one mechanism), the proposal should allow that:

13. the mechanism is formally described and verified. The system as a whole including both normal and exception handling operating modes should be liable to formal checking analysis.
14. the temporal behaviour of the EHM implementation is as much as possible predictable/controlled. In real-time systems, execution time of the EHM part of an application should be taken into account when calculating temporal properties of execution scenarios.

1.2 Sources of Exceptions in CSP-based Architectures

Within the CSP/CT architecture, exceptional events may be expected to occur in the following different contexts:

1. run-time environment:
    a. Run-time libraries and OS – illegal memory address, memory allocation problems, division by zero, overflow, etc…
    b. CT library components can raise exceptions (e.g. network device drivers or remote link drivers on expired timeout; array index outside the range, dereferencing a null pointer).
2. invalid(ated) channels (i.e. broken communication link, malfunctioning device or "poisoned" channels).
3. consistency checks inserted at certain places in a program can fail (e.g. a variable can go outside a permitted range).
4. exceptions induced by exceptions raised in some of the processes important to the execution of the process.

1.3 Mechanism of Exception Propagation

After being thrown, an exception propagates to the place it can be eventually caught (and handled). A crucial mechanism of an exception handling facility is its propagation mechanism, which determines how to find a proper exception handler for the type of exception that has been thrown. Exception propagation always follows a hierarchical path, and in languages different choices are made [15, 16, 18]: dynamically along the function call chain or object creation chain or statically along the lexical hierarchy [19]. The exception propagation mechanism is crucial in understanding the execution flow in presence of exceptions and its complexity directly influences acceptance of the concept in practice.

1.4 Termination and Resumption EHM Models

Occurrence of an exception causes interruption of the ordinary program flow and transfer of control to an exception handler. The state of the exceptionally interrupted processes is also a concern.

Depending on the flow of execution between the ordinary and exceptional operation of software (in presence of an exception), the so-called *handling models* [15] can be predominantly divided in two groups: *termination* and *resumption* EHM models.

In the termination model, further execution of an "exception-guarded" process, function or code block interrupted by an exceptional occurrence is aborted and never resumed. Instead, it is the responsibility of the exception handler to bring the system in such a state that it can continue providing the originally specified (or gracefully degraded) service. If the exception handler is not capable of providing such a service, it will throw the exception further. Therefore, adopting the termination model has intrinsically an unwelcome feature: the functionality of the interrupted process after the exceptional occurrence (termination) point has to be repeated in the handler. It may easily happen that the entire job before the exception occurrence has to be repeated. Therefore, the idea of allowing (also) the resumption mechanism within an EHM does not lose any of its attractions.

In the resumption model, an exception handler will also be executed following the exception occurrence; however, the context of the exceptionally interrupted process will be preserved and after the exception is handled (i.e. the handler terminated), the process will continue its execution at the same point where it was interrupted.

Both exception handling models gained initially equal attention, but practice made the *termination model* prevail for sequential EHMs, as much simpler to implement. It is adopted in all mainstream languages, as C++, Java and Ada.

2 Exception Handling Facilities in CSP-based Architectures

The EHM models discussed in the next section are to address the concurrency-specific issues and therefore *aimed to be used at the level of processes in a process-oriented concurrent environment*. They should be implementable in any language suitable for implementing the CSP principles themselves.

It is another wish that the mechanism does not restrict use of sequential exception handling facilities (if any) present in a chosen implementation language. If a process encapsulates a complex algorithm that is originally developed with use of some native exception handling facilities, there should be no need to modify the original code. As long as the use of a native EHM is confined to *internal* use within a process, it does not clash with the EHM on the process-level. Practically, this means that internally used exceptions *must all be handled within the process*. However, as the last resort, a component should submit all unhandled exceptions to the process-level EHM complying with the process-level exception handling mechanism.

The principal difficulty with concerting error recovery in concurrent systems is posed by the fact that an exception occurrence in one process is an asynchronous event with respect to other processes. In a system designed as a parallel composition of many processes, proper handling of an exception occurrence that takes place in one of the participating processes might require that other dependent processes are interrupted as well.

Propagation of unhandled exceptions is performed according to the hierarchical structure of exception handlers. In occam and the CSP/CT framework, the system is structured as a tree-like hierarchy made of constructs as branches and custom user processes containing only channel communications and pure computation blocks as leaves. A natural choice is to reuse an existing hierarchical construct/process structure and to use processes and constructs as basic exception handling units. This choice can be implemented in few ways:

- every process/construct can be associated with an exception handler,
- extended, exception-aware versions of processes/constructs can be used instead of ordinary processes and constructs,
- a particular exception handling construct may be introduced.

Regardless any particular implementation, upon an unsuccessful exception handing at the process level, the exception will be thrown further to the scope of a construct.

Due to implementation issues, the termination model is preferred at the leaf-process level in an application. The termination model applied at the construct level would mean that prior to the execution of a construct-level exception handler all the subprocesses of the construct would have to terminate. This can happen in several ways: one can choose to wait till all subprocesses terminate (regularly or exceptionally) or force aborting further execution of all subprocesses. In real-time systems where timely reaction to unexpected events is very important the latter may be an appropriate choice. Abandoning the termination model (at the construct level) and implementing the resumption model is a better option when an exception does not influence some subprocesses at all or influences them in a way that can be handled without aborting the subprocesses. Using the resumption model at the construct level would not imply that a whole construct has to be aborted in order to handle the exception that propagated to the construct level.

2.1 Asynchronous Transfer of Control (ATC)

One way to implement the termination model is by an internal mechanism related to the constructs that can force the execution environment to abort all subprocesses and release all the resources they might be holding. This approach resembles Ada's ATC – Asynchronous

Transfer of Control or asynchronous notification in Real-time Java. However forcing exceptional termination of all communicating, parallel composed processes poses a higher risk of corrupting process states by an asynchronous abortion (therefore in the Ada Ravenscar Profile [20] for high-integrity systems, the ATC is disabled). It is important to state that such a mechanism should be made in a way that all aborted subprocesses are given chance to finish in a proper state. This can be done by executing the associated exception handlers for each subprocess.

2.2 Channel Poisoning

The other, more graceful, termination model is *channel poisoning*; sending a poison (or reset) along channels in a CSP network is proposed in [21] as a mechanism for terminating (or resetting) an occam network of processes. Processes that receive the poison spread it further via all the channels they are connected to. Eventually all processes interconnected via channels will receive the poison token and terminate. The method can be used for implementing the termination model of constructs. In the CSP/CT framework this approach is slightly modified as proposed in [2]: instead of passing the poison via the channels, the idea is to poison (invalidate) the channels. Furthermore, in [9], it is proposed that any attempt to access a poisoned channel by invoking its read/write operations will result in throwing an exceptions in the context of the invoking process. Consequently the exception handler associated with the process can handle the situation and/or poison other channels.

3 Architectures of EHM Models

Having in mind all the challenges for constructing a usable EHM for concurrent software, the CSP architecture can be viewed as one offering an interesting environment for doing that. In this part a few concepts are discussed with one eye on all the listed requirements, among which a special concern is given to: handling of simultaneous exceptions, the mechanism formalization and (timely) efficient implementation.

3.1 Formal Backgrounds of EHM

The first CSP construction that captures the behaviour when a process (`Q`) takes over after another process (`P`) signals a failure is conceived by Hoare already in 1973 [22] as

```
P otherwise Q.
```

Association of a process and its handler can be modelled as in **Figure 1**.

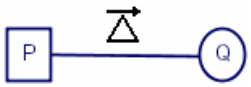

Figure 1. Exception relation between a process `P` and its exception handling process `Q`

In the graphical notation as implemented in the graphical gCSP tool [8], the exception handling process `Q` (exception handling processes are represented as ellipses) is associated with the exception-guarded process `P` (ordinary processes are rectangles) by a compositional *exception relationship* [2], following actually Hoare's "otherwise" principle. On the similar grounds, there have been several attempts to use CSP to formalize exception handling [2, 17, 23, 24]. However, all these attempts have been limited to formalizing the basic flow of activity upon exceptional termination of one process for benefit of another

(thus without building a comprehensive mechanism that fulfils aforesaid requirements for a concurrent EHM). Also, they did not work out an implementation in a practical programming language (with exception of [2]). Common for all is that both ordinary operation and exceptional operation are encapsulated in processes. The compositionality of the design is preserved by combining these processes by a construct.

Hoare eventually also catered for the basic termination principle with his *interrupt operator* (Δ) in [25], in a follow-up work [26] annotated with an exception event `i` (Δ_i). Despite its name, semantics of the Δ operator is much closer to the termination model of exception handling than to what is today usually referred to as "interrupt handling", since it implies termination of the left hand-side operand by an unconditional preemption by the right hand-side one. A true "interrupt" operator would be useful for modelling the resumption model of exception handling (as it actually was in the original proposal of the *interrupt* operator in [23]). In [25], yet another operator (*alternation*, depicted as ⊗) may be used to describe resuming a process execution after execution of another process (however, this operator is not supported by the FDR model checker, while Δ is). In a recent work [27] a CSP-based algebra (with another variant of the Hoare's exception-interrupt constructs) is developed for long transactions threatened by exceptional events. The handling of interrupts (exceptions) relies on the assumption that *compensation* for a wrongly taken act is always possible. This assumption is too strong in the context of controlling mechanical systems (with ever present real-time demands). Moreover, the concept focuses on undoing wrong steps and not directly on fault tolerance.

Termination semantics is captured, besides Hoare's Δ, also by (virtually the same) *except* proposed in [23] and *exception operator* $\vec{\Delta}$ that appears in [2]. Whichever version is used for modelling the exceptional termination of a process `P` that gets preempted by the handler `Q`, it can be represented by a compositional hierarchy (in **Figure 2**) that corresponds to **Figure 1** as:

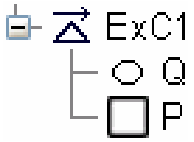

Figure 2. Compositional hierarchy of an exception construction

By "compositional hierarchy" we assume the way occam networks are built of processes and constructs (which are also processes). We find the tree structure excellently capturing this kind of executional compositions [8].

3.2 Exception Construct $\vec{\Delta}$

In the semantics of the $\vec{\Delta}$ operator [2], the composition in **Figure 2** is interpreted as following: upon an exception occurrence in process `P`, an exception is thrown and `P` terminates; the exception is being caught by the exception construct (`ExC1`) and forwarded to `Q` which begins its execution (handling the exception).

A concept of using a construct for modelling exception handling has a favourable consequence to the mechanism of propagating (unhandled) exceptions: in a CSP network with exception constructs, from the moment an exception is created and thrown by a process, it propagates upwards along the compositional hierarchy until a proper handler is found. Therefore, the propagation mechanism is clear and simple, since it follows the compositional structure of the CSP/CT concurrent design.

Instead of the process `P` in Figure 2, there may be a construct with multiple processes. If the construct is an Alternative or Sequential one, the situation is the same as with a single

process: upon exceptional termination of one of the alternatively or sequentially composed processes, the exception is caught by the exception construct and handled by the process `Q`. However, in the case of the Parallel construct, there is a possibility that more than one process ends up in an exceptional situation (and therefore terminates by throwing different exceptions). Consider situation in Figure 3.

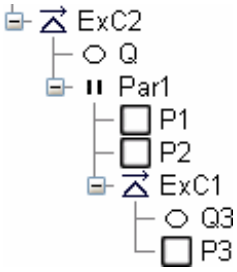

Figure 3. Parallel construct under exception construct

Handler `Q` handles exceptions that may arise during execution of the parallel composition of the processes `P1`, `P2` and the exception construct `ExC1` (actually, the exceptions thrown by `P3` and not handled by `Q3`). Here, it is a question at which moment exceptions from `P1` should be handled (provided that the exception occurrence happens before `P2` finishes)? Moreover, what if `P2` exceptionally terminates as well?

In the current implementation of $\vec{\Lambda}$ [2, 28], the exceptions occurred in parallel composed processes are handled when the Parallel construct is terminated (i.e. when all parallel composed processes are terminated, successfully or exceptionally); for catching and handling all possible exceptions occurred in a parallel composition, a concept of *exception set* (collection of exceptions) is introduced. After termination of `Par1`, handler `Q` gets an exception set object with all exceptions thrown by the child processes (`P1`, `P2` and `ExC1` – all possibly unhandled processes in `Q3` are rethrown).

The concept of exception set has another useful role. From its contents a handler can reconstruct the exception hierarchy in case of simultaneous (concurrent) exceptions.

3.2.1 Channel Poisoning and the Exception Construct

Sending a poison along channels as proposed in [21] is a mechanism for terminating the network or subnetwork. In the discussed EHM model proposal the poisoning mechanism assumes that channels can be turned into a poisoned state in which they respond on attempts of writing or reading by throwing back exceptions. In this way two problems are solved.

The first problem is blocking of a rendezvous partner when the other one has exceptionally terminated. Consider the following situation (**Figure 4**, **Figure 5**):

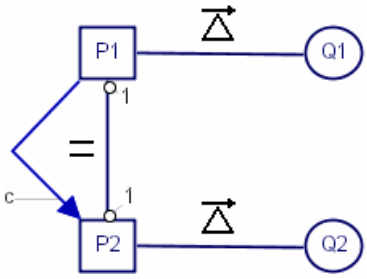

Figure 4. Rendezvous (potential blocking)

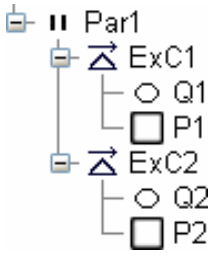

Figure 5. Hierarchical representation of Figure 4

Processes `P1` and `P2` are both "exception-guarded" by exception constructs `ExC1` and `ExC2` (i.e. by handlers `Q1` and `Q2` respectively), which are then parallel composed. Processes `P1` and `P2` communicate over channel `c`. Should it happen than one of the processes exceptionally terminates (before the rendezvous point), the other process stays blocked on channel `c`. For that reason, handlers `Q1` and `Q2` should in principle turn the channel into the poisoned state, so that the other party terminates with the same exception which caused the first process to terminate. To recall, this exception is thrown on an attempt of reading or writing. Moreover, an already poisoned channel on further poisoning attempts (which are function calls) returns the poisoning exception, for a reason that will be explained soon. If however the other rendezvous partner is already blocked on the channel, it should be released at the act of poisoning (and then end up with the exception).

For this scheme to work, it is clear that all communicating parallel composed processes should be "exception-guarded", i.e. sheltered behind exception constructs. In that case, an elegant possibility for concerted simultaneous exception handling comes automatically. On an exceptional occurrence in one of the communicating processes, provided all are accompanied with handlers that poison all channels connected to "their" processes, the information of the exception spreads within the parallel composition. In case of simultaneous exceptions, the spread of different exceptions progresses from different places (processes) under a parallel construct. In that case, it will inevitably happen that a handler will try to poison a channel that is already poisoned (with another exception). When channels respond to the attempt of poisoning by returning the exception that poisoned them initially, the handlers get information on occurrence of simultaneous exceptions. The handler of the parallel construct will ultimately be able to reconstruct the complete exception hierarchy.

However, this mechanism suffers from two major problems. The first one is possible (unbounded) delay from occurrence of a (first) exception and handling it at the level of the parallel construct. Remember that all parallel composed processes must terminate before the handler of the parallel construct gets chance to analyse and handle the exception (set). Some processes may spend a lot of time before coming to the rendezvous on a (poisoned) channel and consequently be terminated! A possibility of Asynchronous Transfer of Control is already commented as unwelcome in the high-integrity systems. An additional penalty is that for the mechanism of rethrowing exceptions from poisoned channels to work, it is necessary to clone exceptions (so that every handler can consider the total exceptional situation) or at least have a rigorous administration of the (pointers to) occurred exceptions.

The other problem inherent to the mechanism of channel poisoning is that a poison spread is naturally bounded by the interconnection network of channels and *not* by the boundaries of constructs. Namely, some channels may run to processes that belong to other constructs; ultimately this may lead to termination of the whole application, which contradicts to the idea of exception handling as the most powerful fault tolerance mechanism. In [21] a possibility of inserting special processes on boundaries of the subnetworks compelled to poisoning is proposed, but that means introducing completely non-functional components into the system. The other option is a model-based (tool-based) control of the poison spreading.

3.3 Interrupt Operator Δ_i, Environmental Exception Manager and Exception Channels

In the channel poisoning concept, propagation of an exception *event* was based on existing communication channels. More apt to formal modelling would be a termination model based on a concept that considers exceptions as explicit events communicated among

exception handlers via explicit *exception channels*. This change in paradigm makes formal modelling and checking more straightforward.

Let us consider a Parallel construct `Par` containing 3 subprocesses: `P1`, `P2` and `P3` (see Figure 6). Process-level exception handlers associated with these processes are `Q1`, `Q2` and `Q3` respectively. In the scope of the exception construct `ExC` the exception handler associated with the construct `Par` is process `Q`.

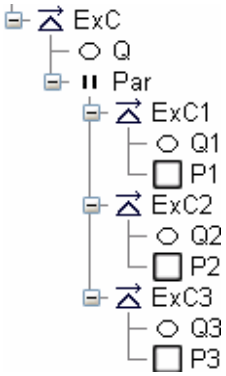

Figure 6. Design rule for fault-tolerant parallel composition with environmental care

Using the interrupt operator this could be written in CSP as (i_n is an explicit exception event):

$$Par = (P1\ \Delta_{i1}\ Q1)\ ||\ (P2\ \Delta_{i2}\ Q2)\ ||\ (P3\ \Delta_{i3}\ Q3).$$

In turn, the relation between processes `Par` and `Q` is modelled in the same way:

$$Par\ \Delta_i\ Q = ((P1\ \Delta_{i1}\ Q1)\ ||\ (P2\ \Delta_{i2}\ Q2)\ ||\ (P3\ \Delta_{i3}\ Q3))\ \Delta_i\ Q,$$

where `i` is the `Par`-level exception event.

If an exception occurs during execution of a process `P1`, the process will be aborted and the associated exception handler `Q1` will be invoked. This can actually be seen as an implicit occurrence of the exception event `i1`. If the exception cannot be handled by `Q1`, this exception should be communicated to the higher EHM level. Since such higher level EHM facilities are represented by some process from the environment playing a role of a higher-level exception handler, this can be implemented as communication via channels. One can imagine that, following premature termination of a process, a higher EHM component can throw exceptions in the contexts of the other affected processes. In sense of CSP, this is equal to interrupting those processes, by inducing an event `i2` that will cause, say process `P2` to be aborted and wake up its exception handler (`Q2`). In this way, graceful termination (giving a chance for a process state clean-up) can be modelled by the CSP-standard interrupt operator Δ_i.

Thus, from the interrupt operator Δ_i point of view aborting a process (`P2`) is nothing more then communicating the exception event (`i2`) to the exception handling process (`Q2`). And indeed the termination mechanism can be really implemented in this way. Special exception channels can be dedicated to this purpose. The communication via exception channel is actually an encapsulating mechanism used to throw an exception in the context of affected processes (`P2` and `P3`), forcing them to abort further execution and forcing the execution of associated exception handlers (`Q2` and/or `Q3`) instead.

Although their implementation is more complicated, from synchronization point of view those channels are real rendezvous channels. This is the case because processes `Q1`, `Q2` and `Q3` are during ordinary operation mode always ready to accept events `i1`, `i2` and `i3` produced by the environment.

Writing to an exception channel would pass data about the cause of the exception to the process-level exception handler. In addition, the process must be unblocked if it is waiting on a channel or semaphore. Afterwards, when a scheduler grants CPU time to that process, instead of a regular context switch to the stack of the process, it would switch to the stack unwrapped to a proper point for the execution of the exception handler.

When all process-level handlers `Q1`, `Q2` and `Q3` terminate, the construct (`Par`) will terminate unsuccessfully by throwing an exception to its parent exception construct (`ExC`). As a consequence, the exception handler `Q` will be executed.

But who will produce events `i1`, `i2` and `i3`? The exception handler `Q` cannot do that because it can be executed only after the construct and all of its subprocesses have already terminated. It is possible to imagine an additional environment process (let us name it *environmental exception manager - EEM*) that does that. This process would have to run in parallel with the guarded construct or the whole application. Furthermore, because the exception handling response time is important, this newly introduced process should have a higher priority then the top application construct. For the running example,

$$\texttt{PriPar (EEM, Par } \Delta_{\texttt{i}} \texttt{ Q).}$$

Every process is by default equipped with an exception handler which, if not redefined by the user, only throws all exceptions further to the environmental exception manager. While for the previous concept it was not necessary that all processes have associated handlers in order to let their exception be handled at the construct level, in this proposal it is the rule all processes must have attached handlers (as in Figure 6). One side-effect of this decision is that it becomes possible to define both a process and its exception handler as two functions of one object. Normally, in the occam-like libraries, processes are implemented as objects, but this was just a design choice since from the CSP point of view there is no obstacle in realizing a process as merely a function. While a process and its exception handler were defined in separate objects, the process had to pack all the data needed for exception handling into an exception object in order to pass it to its exception handler. Having them inside the same object is however more convenient in light of real-time systems. Besides reducing the memory usage, the dynamic memory allocation can be avoided since an exception handler can directly inspect data members defining the state of the process.

The concept of the exception manager opens yet another possibility: thanks to the careful management of the exceptions events, the resumption feature becomes viable. The manager can have encoded application-specific exception handling rules. These rules may not necessarily terminate all subprocesses. The termination and the resumption model can be combined in one application.

3.3.1 Treating Complex Exceptional Situations

In order to make an appropriate handling decision about occurrence of simultaneous exceptions in multiple processes (sometimes caused by the same physical fault), it is often necessary to check the state of certain resources internal to concurrently executing constructs. Obviously, handling such complex exception events requires some kind of exception hierarchy checks and application specific rules that are encoded for all possible combinations. If the number of those rules and combinations is very large, which is the case in complex systems, the environmental exception manager can be implemented as a complex process containing several *environmental exception handlers* covering different functional views of the system or different classes of exceptional scenarios. It is also possible to create one environmental exception handler for every construct in the system.

All environmental exception handlers are processes. As such they can communicate with each other and use the knowledge obtained in such a way to decide whether they should safely terminate certain subprocesses by producing exception events.

Each (Parallel) construct can have an associated higher level environmental exception handler. One environmental exception handler can handle one or more constructs. If the termination model is used, then this exception handler will throw an exception to every subprocess of the construct in order to terminate them in a safe way. The termination model should be a default behaviour. But if one needs the resumption model, a behaviour of such a higher-level environmental exception handler can be adapted and a set of application specific rules can be coded in order to specify how to deal with exceptions present in that case.

In this way, by interpreting all exceptions as events, the whole application, including exceptional situations, can be formalized and formally checked via CSP model. The time from raising an exception inside a process-level exception handler until reaction by the environmental exception manager is bounded and there is no need to terminate processes not influenced by exceptions. Even if the resumption is not used, exception handling takes place immediately without a time delay and the termination can be much more efficiently accomplished. Another benefit is that simultaneous and related exceptions that originate in different constructs can be treated properly. Collecting exceptions in the exception sets is not used in this concept, since the environmental exception manager maps a subtree of the exception hierarchy to an appropriate exception event used for activating the Δ_i-mechanism at the construct level. The problem of unwanted uncontrolled exception propagation beyond construct boundaries does not exist at all.

4 Conclusions

Exception handling facilities are a crucial tool for increasing dependability level of the software in an elegant manner. However, “exception handling and the provision of error recovery are extremely difficult in concurrent and distributed systems” [18]. Nevertheless, we find process-oriented architectures based on CSP model very promising for facilitating a concurrent exception handling mechanism.

In subsections 3.2 and 3.3 two concurrent EHMs based on constructs (the exception construct $\vec{\Delta}$ from [2] and Δ_i from [25] and [26]) are proposed. The first one has been successfully implemented and demonstrated in robotic applications [28]; the other is currently under development. While the first one facilitates the termination model only, the second proposal holds promise of providing the resumption possibility as well.

Especially attractive is the potential to have programs with exception handling facilities formally verified. Both exception operator and interrupt operator are useful starting points for designing a formally checkable EHM. For both proposals in this paper formal description and verification have to be yet exercised.

Both EHMs are based on introduction of constructs that fit them nicely in the CSP hierarchical compositions, which simplifies reasoning and offers excellent possibilities for graphical specification in the graphical language and tool [8]. Extending the tool support towards automatic code generation and consistency administration is active research [9].

References

[1] C. A. R. Hoare, Communicating Sequential Processes, *Communications of the ACM*, vol. 21, pp. 666-677, 1978.

[2] G. H. Hilderink, *Managing Complexity of Control Software through Concurrency,* PhD thesis, University of Twente, Netherlands, 2005, ISBN: 90-365-2204-8.

[3] G. H. Hilderink, JavaPP project at UT: `http://www.ce.utwente.nl/JavaPP` `http://www.ce.utwente.nl/JavaPP`, 2002.

[4] B. Orlic and J. F. Broenink, Real-time and fault tolerance in distributed control software, in *Communicating Process Architectures 2003*, J. F. Broenink and G. H. Hilderink, Eds. Enschede, Netherlands: IOS Press, 2003, pp. 235-250, ISBN: 1 58603 381 6.

[5] P. H. Welch, Process Oriented Design for Java: Concurrency for All, in *ICCS 2002*, volume 2330 of Lecture Notes in Computer Science. Amsterdam: Springer-Verlag, 2002, pp. 687-687, ISBN: 3-540-43593-X.

[6] J. Moores, CCSP – A portable CSP-based run-time system supporting C and occam, in *Architectures, Languages and Techniques – WoTUG-22*, B. M. Cook, Ed. Keele, UK: IOS Press, 1999, pp. 147-168, ISBN: 90 5199 480 X.

[7] N. C. C. Brown and P. H. Welch, An Introduction to the Kent C++CSP Library, in *Communicating Process Architectures 2003*, J. F. Broenink and G. H. Hilderink, Eds. Enschede, Netherlands: IOS Press, 2003, pp. 139-156, ISBN: 1 58603 381 6.

[8] D. S. Jovanovic, B. Orlic, G. K. Liet, and J. F. Broenink, gCSP: A Graphical Tool for Designing CSP systems, in *Communicating Process Architectures 2004*, I. East, J. Martin, P. H. Welch, D. Duce, and M. Green, Eds. Oxford, UK: IOS press, 2004, pp. 233-251, ISBN: 1586034588.

[9] D. S. Jovanovic, *Designing dependable process-oriented software, a CSP approach,* PhD thesis, PhD thesis, University of Twente, NL, to appear in 2005.

[10] A. Romanovsky, Ed. *Looking ahead in atomic actions with exception handling*.

[11] T. Anderson and P. A. Lee, *Fault Tolerance, Principles and Practice*. Englewood Cliffs, NJ: Prentice-Hall International, 1981.

[12] J. B. Goodenough, Exception Handling: Issues and A Proposed Notation, *Comm. ACM*, vol. 18, pp. 683-696, 1975.

[13] R. H. Campbell and B. Randell, Error Recovery in Asynchronous Systems, *IEEE Trans. Software Eng.*, vol. 12, pp. 811-826, 1986.

[14] F. Cristian, Exception Handling and Tolerance of Software Faults, in *Software Fault Tolerance*, vol. 3, *Trends in Software*, M. R. Lyu, Ed., 1 ed. Chichester: John Wiley & Sons Ltd., 1995, pp. 81-107.

[15] P. A. Buhr and W. Y. R. Mok, Advanced Exception Handling Mechanisms, *IEEE trans. Software Eng.*, vol. 26, pp. 820-836, 2000.

[16] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, 3rd ed: Pearson Education, 2001.

[17] J.-P. Banatre and V. Issarny, Exception handling in communicating sequential processes: design, verification and implementation, INRIA, France, Rennes RR-1710, June 1992.

[18] J. Xu, A. Romanovsky, and B. Randell, Concurrent exception handling and resolution in distributed object systems, *IEEE Trans. on Parallel and Distributed Systems*, vol. 11, pp. 1019-1032, 2000.

[19] J. L. Knudsen, Exception Handling – A Static Approach, *Software – Practice and Experience*, vol. 14, pp. 429-449, 1984.

[20] A. Burns, The Ravenscar Profile, *ACM Ada Letters*, vol. XIX, pp. 49-52, 1999.

[21] P. H. Welch, Graceful Termination – Graceful Resetting, in *occam User Group X*. Enschede, Netherlands: IOS Press, 1989, pp. 310-317, ISBN: 90 5199 007 3.

[22] C. A. R. Hoare, Parallel programming: an axiomatic approach, Stanford University, Stanford, CA, USA CS-TR-73-394, October 1973.

[23] T. I. Dix, Exceptions and Interrupts in CSP, *Science of Computer Programming*, vol. 3, pp. 189-204, 1983.

[24] P. Jalote and R. H. Campbell, Fault tolerance using Communicating Sequential Processes, in *International Symposium on Fault-Tolerant Computing, FTCS-14.*, 1984, pp. 347 - 352.

[25] C. A. R. Hoare, *Communicating Sequential Processes*: Prentice Hall, 1985.

[26] A. W. Roscoe, *The Theory and Practice of Concurrency*: Prentice Hall, 1997.

[27] M. Butler, C. A. R. Hoare, and C. Ferreira, A Trace Semantics for Long-Running Transactions, in *Communicating Sequential Processes – The First 25 Years*, vol. 3525, *LNCS*, A. E. Abdallah, C. B. Jones, and J. W. Sanders, Eds. Berlin Heidelberg: Springer-Verlag, 2005, pp. 133-150.

[28] T. H. van Engelen, CTC++ enhancements towards fault tolerance and RTAI, Control Laboratory, University of Twente, Enschede, MSc thesis 022CE2004, 2004.