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

Exception Handling Mechanism in Communicating Threads for Java

Gerald. H. HILDERINK
Boulevard 1945 – 139, 7500 AE, Enschede, The Netherlands
g.h.hilderink@zonnet.nl

Abstract. The concept of exception handling is important for building reliable software. An exception construct is proposed in this paper, which implements an exception handling mechanism that is suitable for concurrent software architectures. The aim of this exception construct is to bring exception handling to a high-level of abstraction such that exception handling does scale well with the complexity of the system. This is why the exception construct supports a CSP-based software design approach. The proposed exception construct embraces informal semantics, but which are intuitive and suitable to software engineering. The exception construct is prototyped in the CSP for Java library, called CTJ.

Keywords. Exception handling, CSP, concurrency, real-time, embedded software.

Introduction

Reliable software should deal with *all* circumstances in its environment, which can affect the behaviour of the program. The environment of the program encompasses the computer hardware. Unusual circumstances are exceptional occurrences that can bring the program, when not dealt with, in a state of undesirable behaviour. This causes an exceptional state, which manifests an error or simply an *exception*. The processes in the program that are affected by the exception should not progress after the occurrence of the exception. Each process that is encountering an exception should escape to a handler process that is able to deal with the exception. This handler process is called an *exception handler*.

Reasoning about the behaviour of the program in the presence of exceptions can be very complex. Branching to an exception handler can occur at many places in the program. Exceptions occurring in exception handlers require branching from exception handler to exception handler. Exceptions are related to the concurrent behaviour of the system. Exceptions can occur asynchronously and simultaneously in concurrent systems. There can be too many states to consider when tracing the behaviour of exception handling.

Exceptions should be handled by proper design concepts that deal with its complexities. A proper concurrency model is inevitable in order to manage the complexity of exception handling. Proper design concepts can be found in the CSP concurrency model. The CSP concepts provide sufficient abstraction, compositionality and sound semantics that are very suitable for designing and implementing mission-critical embedded software. However, CSP does not specify a simple solution for describing exception handling.

An informal description of an exception construct is presented, which offers a simple solution to handle exceptions in concurrent software architectures. On one hand, the approach is in accordance with CSP terminology. A formal description and analysis in CSP is not part of this paper. The feasibility of the exception construct has been investigated. The exception constructs has been prototyped in the *Communicating Threads for Java*

(CTJ) library [1; 2]. On the other hand, the approach is based on a software engineering perspective.

The notion of exceptions is discussed in Section 1. This notion follows the CSP terminology. The role of the environment of the program, poisoning channels and processes are discussed. The concept of exception handling is discussed in Section 2. An example program with nested exception constructs is described in Section 3. Various aspects of the exception construct are discussed in Section 5. Section 5 deals with the conclusions.

1. Exceptions

1.1. Processes, Events, and Channels

An elegant way to design an implement mission-critical software in embedded systems is the use of *Communicating Sequential Processes* (CSP) concepts [3; 4]. CSP is a theory of programming, which offers formal concepts for describing and reasoning about the behaviour of concurrent systems. Furthermore, CSP offers pragmatic concepts and guidelines for developing reliable and robust concurrent process architectures. These concepts are process-oriented and they offer abstraction, compositionality, separation of concerns and a concurrency model with simple and clean semantics.

CSP is a notation for describing concurrent systems by means of processes and events. Processes are self-contained entities, which are defined in terms of events. Processes do not see each other, but they interact with other processes via channels. An event is an occurrence in time and space. An event represents a mutual synchronisation between two or more processes. A process that is willing to engage in an event must wait until other processes are also willing to engage in the event. This is called the *rendezvous* or *handshake principle*.

The rendezvous between two processes that are willing to communicate via a shared channel is called the *communication event*. A process that successfully terminates engages in a *termination event* with its subsequent process.

1.2. Process in Exception

An exception is a state in which

a) An instruction is causing an error and the instruction cannot complete or successfully terminate; e.g. division by zero or illegal address.

b) A communication event is refused by the environment of the program; e.g. the channel implementation is malfunctioning.

In either case, the environment in which the program runs cannot let a process continue after the point of exception. A *process in exception* will never engage in any event after the exception has been raised in the process; i.e. it behaves as *STOP*. Furthermore, a process is in exception when at least one of its sub-processes is in exception.

Conceptually, the exception handling mechanism interrupts the process in exception and it will be replaced by the exception handler. If the exception handler terminates, the exception handling process terminates normally.

1.3. The Role of the Environment

The role of the environment in the exception handling mechanism is important in order to understand the source of exceptions. For some reason it could happen that a device in hardware (the environment of the program) is malfunctioning and it cannot establish or

complete communication. In other words, the communication event is refused by the environment of the program. The event will never occur and the process may wait forever for the event to happen. This could cause the program to deadlock or livelock. This is an inconvenient circumstance, which manifests an exception. It is more convenient for the program to escape from the exception and to do something useful; e.g. dealing with the exception.

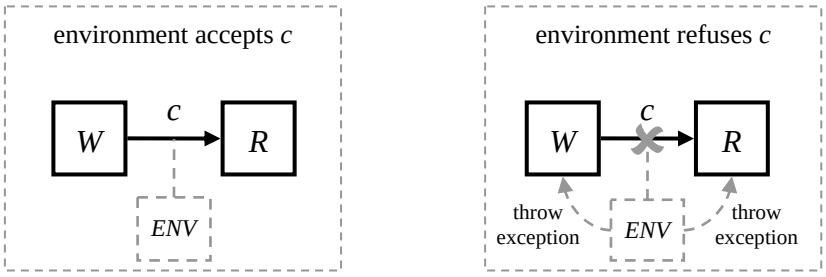
The environment of a program is usually something complex from which the software engineer wants to abstract away from. The software engineer is interested in the causality between a misbehaving environment and the behaviour of the program. The CSP channel model supports this view.

Figure 1 illustrates two parallel processes communicating via channel *c*. The figure is called a *CSP diagram* [1]. The writer process *W* writes data to the channel and the reader process *R* reads the data from the channel. The figure does not show exactly when they communicate. The points of communication will be illustrated in Section 2.2. During the design of a program, the environment is not included in the design, but its effect on the design should be considered.

(a) Communication diagram (b) Composition diagram

Figure 1. Two parallel processes communicate via a channel (CSP diagram).

The abstract role of the environment is illustrated in Figure 2. Figure 2aillustrates that the environment can be depicted as a parallel process, named *ENV*. The environmental process *ENV* is listening on channel *c* and it decides whether or not to participate in the communication event. *ENV* is dotted to illustrate that the environmental process is hidden in the design, but it is an integral part of the implementation of the design. In fact, the channel implementation can be viewed as an environmental process, because the channel implementation directly controls the underlying hardware. The interface of the channel separates the program from its environment. The processes in the program should only access the devices via channels. This abstraction and separation of concern keeps the processes free from hardware depending code. Of course, this hardware-independency excludes the integrity of the processes, i.e. processes depend on the data provided by the channels (or devices).



(a) Environment is functioning properly (b) Environment is malfunctioning

Figure 2. The role of the environmental process.

In case the channel *c* breaks, the environment will refuse *c* from happening. This is illustrated in Figure 2b. Instead, the channel will throw (or raise) an exception to each

involved process. The grey arrows indicate the source and destination of throwing exceptions from the channel implementation to the invoking processes.

The channel is modelled as an active partner process in its communications, that can be put into "refusal" and exception-throwing mode, where "exceptions" are just events for which other processes have (CSP) interrupt handlers awaiting.

1.4. Poisoning Channels and Processes

A channel being refused by the environment of the program is called a *poisoned channel*. A poisoned channel will never cause a communication event to happen as long as it is poisoned. A poisoned channel devotes its channel ends to throw exceptions to the processes that are willing to communicate via its *poisoned channel ends*. After the exception is thrown, the process is poisoned and it will eventually die; i.e. a *poisoned process* never engages in any event with its environment and it never terminates normally. Successively, the exception will be caught by a surrounding exception construct.

In CTJ, the methods `refuse(channel)`, `refuse(channel,exc)` and `accept(channel)` were introduced [2]. These methods are defined by a static call-channel that is connected to the environmental process *ENV*. Invoking one of the refuse methods will request a "refuse" service from the environmental process *ENV*. After *ENV* accepts the request, any communication event on the specified channel will be refused. In case the exception argument `exc` is specified, the channel will throw `exc` via the channel ends on which processes are willing to perform the read or write methods. For example, invoking `refuse(c,exc)` corresponds to the situation as depicted Figure 2b. We prefer the method name `refuse` rather than `poison`, which is in accordance with the CSP terminology of "refusals". The method `accept(channel)` requests the environmental process to accept communication events on `channel`; i.e. undo the poisoning, if possible.

The `refuse(..)` and `accept(..)` methods are meant to be used by the implementation of channels or by the underlying kernel. The program could use these methods for studying the effects of poisoning channels on the behaviour of the program; i.e. simulating the effect of malfunctioning devices. In all other situations, we do not encourage these methods to be used by the program. Poisoning channels by processes can be error-prone and therefore it should not be encouraged for deliberately killing processes. For example, poisoning channels by an exception handler could cause exceptions to propagate outside the scope of the exception construct. If this is not desired, poisoning channels is not useful.

The C++CSP library [5; 6] uses a different approach, whereby the `poison()` method is part of the channel interface. This is called *stateful poisoning* of channels. A process is being poisoned while attempting to access a poisoned channel must poison all channels it uses before terminating gracefully. Special functions can be used, which provide channel ends that cannot be used to poison the channel. Processes can choose whether the channel ends they pass to their sub-processes can be poisoned or not.

The use of the `refuse(..)` or `poison(..)` methods by processes is error-prone and it will complicate the understanding of the epidemic of poisoning. It is safer to leave the killing (deliberately poisoning) of processes up to the channels and the exception constructs. A process in exception does not need to poison its channel ends. The poisoning of channel ends should be performed by the exception constructs. This mechanism is elaborated in Section 2.

1.5. Termination and Resumption

This exception handling approach encompasses two models of exception handling, namely:

Resumption model. The resumption model allows an exception handler to correct the exception and then return to the point where the exception was thrown. This requires that recovery is possible with acceptable overhead costs. The resumption model is most easily understood by viewing the exception handler as an implicit procedure which is called when the exception is raised. The resumption model is also called retry model [7].

Termination model. In the termination model, control never returns to the point in the program execution where the exception was raised. This will result in the executing process being terminated. The termination model is necessary when error recovery is not possible, or difficult to realize, with acceptable overhead costs. The termination model is also called escape model [7].

Error recovery or resumption is sometimes possible at the level of communication, i.e. by the channels. The channel implementation can detect errors and possibly fix them with if-then-else or try-catch constructs. In case the error is fixed and communication is re-established by the channel, this can be viewed as resumption. In this case, processes are not aware of any exceptions that were fixed. A channel that cannot fix the internal error should escape from resumption. The channel should raise (or throw) an exception via its interface to the process domain. The process domain and the channel domain are depicted in Figure 3.

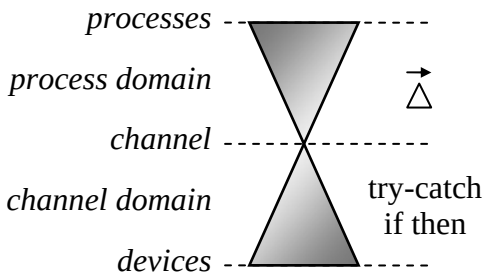


Figure 3. Process and channel domains.

The channel interface separates both domains. The process domain supports the termination model and the channel domain supports the resumption and the termination model. The exception construct resides in the process domain and supports the termination model.

2. Exception handling

2.1. Exception Construct

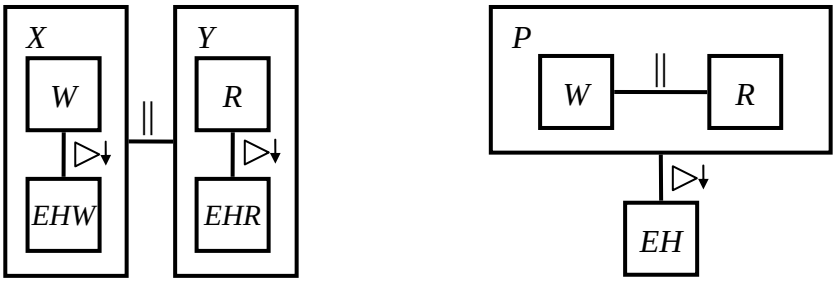
In Hilderink [1], a notation was introduced to describe exception handling in a compositional way. The exception handling is based on an exception construct with a formal graphical syntax, but with informal semantics. The exception construct composes two processes *P* and *EH*, which is written as:

$P \bar{\Delta} EH$ *exception construct*

This process behaves as *EH* when *P* is in exception; otherwise it behaves as *P*. Process *P* is in exception on the occurrence of an error from which *P* must not continue. At the point of exception *P* behaves as *STOP*. Process *EH* is the exception handler.

On the occurrence of an exception, the exception construct requires that all the channel ends, being claimed by *P*, are released. The exception construct must reckon with a complex composition of sub-processes of *P*. The released channel ends can be re-claimed by other processes, for example, by the exception handler *EH*. A poisoned channel end cannot be re-claimed as long as it is poisoned. The exception construct has resemblance with the interrupt operator in CSP. We will omit a theoretical discussion between the formal interrupt operator and the informal exception construct.

Consider the CSP diagram in Figure 1. This example is enhanced with exception constructs. Figure 4 illustrates two different enhancements. The processes are shown transparently. Each compositional diagram depicts a different composition. Figure 4a illustrates the two processes *W* and *R* that are each guarded by an exception construct. Exception handler *EHW* deals with the exception at the writer's side of channel *c* and *EHR* deals with the reader's side of channels *c*. On exception in *c*, the processes *EHW* and *EHR* run in parallel. Figure 4b illustrates the circumstance where the exception handler *EH* deals with both sides of channel *c*. *EH* could be any sequence of *EHW* and *EHR*.



(a) Disjoint exception constructs. (b) Joint exception construct.

Figure 4. Compositional diagrams enhanced with exception handling.

Figure 5 shows an equivalence property between two compositions. The process *SKIP* doesn't do anything, except successfully terminating. Since *SKIP* does not deal with any exception and therefore the exception handler *EH* will take over.

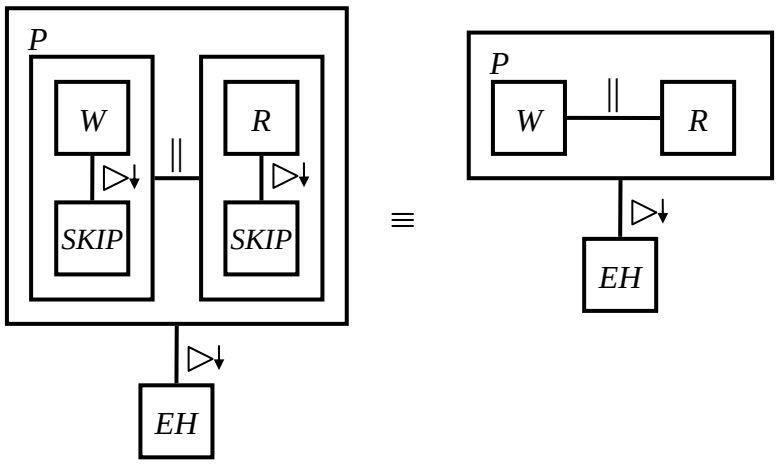


Figure 5. Equivalent exception compositions.

2.2. Exception Handling Mechanism

The conceptual behaviour of the exception handling mechanism of the proposed exception construct is described in this section. The required steps that are performed by the mechanism are explained by a simple example. Furthermore, the channel ends and the scope of the exception construct are explained.

The following steps that are taken by the exception handling mechanism:

1. **Registering.** Register each channel end and nested exception construct, being invoked by a process, to the surrounding exception construct.
2. **Notifying.** Notify the exception construct that an exception has occurred and the exception will be collected by the exception construct.
3. **Poisoning.** Poison the registered channel ends and registered (nested) exception constructs. A poisoned exception construct will propagate its poison. After all, all poisoned channel ends that were claimed by a process will be release.
4. **Throwing.** The channel ends throw NULL exceptions, which exceptions propagate via the CSP constructs until they are caught by the exception construct.
5. **Healing.** Before the exception handler is executed the registered channel ends and registered (nested) exception constructs must be healed. Otherwise these channel ends cannot be re-claimed by the exception handler. Those channel ends that belong to poisoned channels remain poisoned. Those channel ends cannot be re-claimed by the exception handler.
6. **Handling.** The associated exception handler reads the exception set and handles each exception one by one. Exceptions that have been handled by the exception handler must be removed from the set.

Step 1 is performed when no exception has occurred. The steps 2 till 6 are performed on the first occurrence of an exception. Each of these steps is explained in the following example.

The example consists of the processes *U*, *P*, *T* and *EH*. See Figure 6. Process *P* is defined by the processes *R* and *S*. The communication relationships *a*, *b* and *c*, and the compositional relationships are depicted one diagram. The compositional relationships are in grey. Process *P* is related to the exception handler *EH*. The channel inputs and outputs are depicted by primitive reader and writer processes, respectively labelled with '*?*' and '*!*'. These primitive reader and writer processes mark the *channel ends* of the associated channel. In *R* and *S*, the channel ends are related to a sequential composition, which defines: first input, then output.

Each exception construct defines a scope to which a group of channel ends is related. This example illustrates that the channel ends in *P* are in the scope of the nearest exception construct associated with *EH*. The channel ends of *U* and *T* are not within the scope of the exception construct.

The processes *R* and *S* are randomly scheduled on a single processor system. We start with process *R*. Assume *R* is performing the input on the channel *a*. Since the start of *P*, this is the first time this channel end is accessed. On this first access, the channel end is registered to the nearest exception construct. See step ❶ in Figure 7. A second access does not require registering, since the channel end was already registered to an exception construct. Note that each thread keeps a reference to the exception construct to which it is part of. After *S* is scheduled, *S* is willing to input from channel *b*. Also this channel end will be registered to the exception construct in step ❷. Process *S* is waiting for channel *b*.

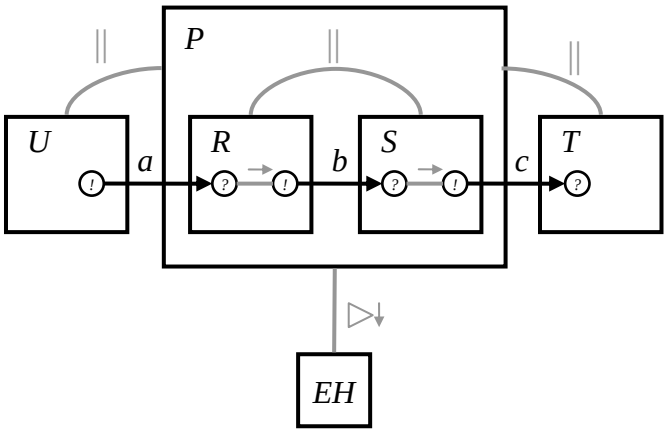


Figure 6. Example of a program consisting of four processes *U*, *P*, *T and EH*.

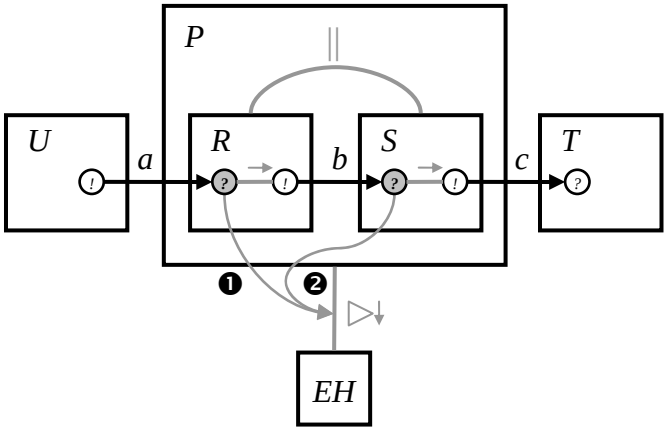


Figure 7. Registering of channel ends to the exception construct.

In the meantime something bad happened with the implementation of channel *b*. After *R* is scheduled and received data from channel *a*, *R* is willing to output on channel *b*. Since the channel *b* is poisoned, its channel ends are also poisoned. Registering of a poisoned channel end is not necessary, which saves at least one registering operation. On the output operation, the channel end will notify the exception construct that an internal exception has occurred. See step ❸ in Figure 8. The exception is collected by the exception construct.

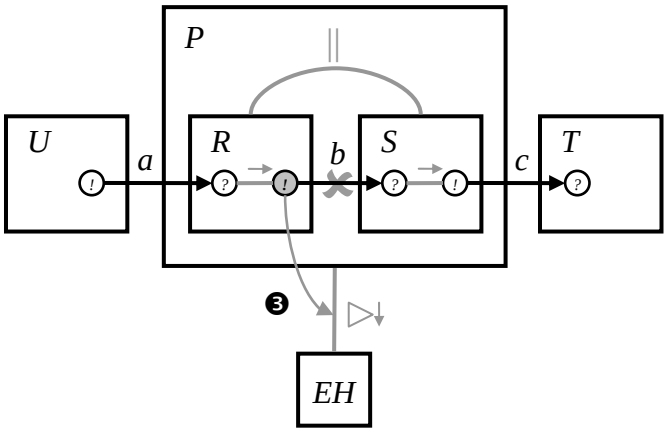


Figure 8. The channel notifies the exception construct that an exception has occurred.

After notifying the occurrence of an exception to the exception construct, the exception construct will immediately poison all registered channel ends. A poisoned channel end will release its synchronization with any process. In this example, the registered channel ends

were the input channel end in *R* and the input channel end in *S*. See step ❹ in Figure 9. The input channel end in *S* is blocking *S* and therefore it will unblock *S*. The input channel end in *R* needs no unblocking, because *R* does no longer claim the channel end of *a*.

The procedure of poisoning the registered channel ends can detect other exceptions in the associated channels. The newly detected exceptions will be collected by the exception construct. It is possible that not all exceptions are detected by this procedure. This is not a problem, since the yet undetected exceptions will be detected at a later time or they will not be detected at all. In the latter case, no harm will be done since these channel ends are never used again.

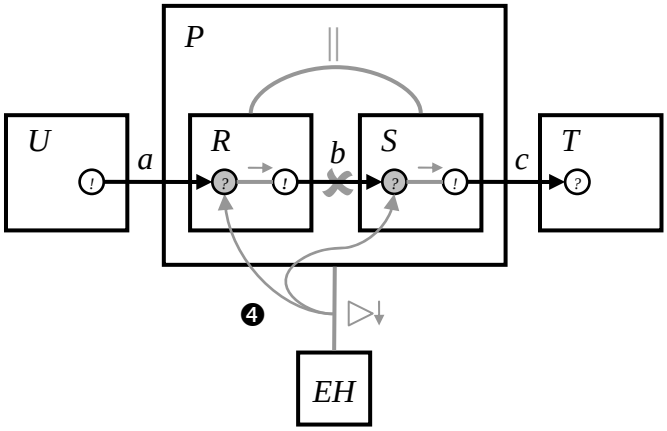


Figure 9. The exception construct poisons the registered channel ends.

The channel ends of a poisoned channel will throw NULL exceptions to each process that accesses the channel end. These exceptions are passed to the hierarchy of compositional constructs until the associated exception construct is reached. See the steps ❺ and ❻ for process *R* and the steps ❼ and ❽ for process *S* in Figure 10.

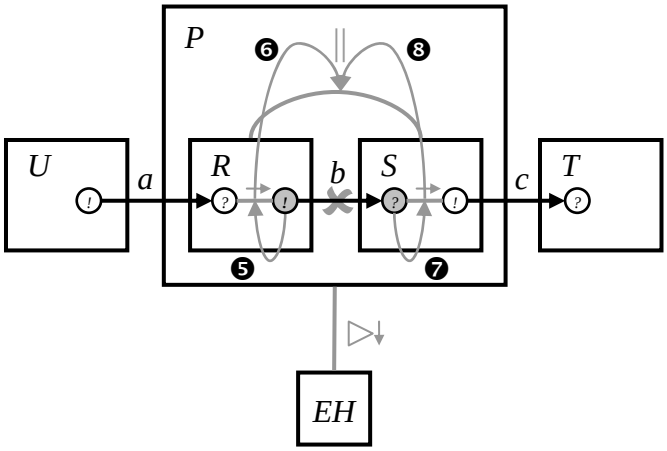


Figure 10. NULL exceptions are thrown from the channel end up the parallel construct.

The NULL exception does not contain information about the actual exception. Note: the actual exception was already collected in step ❸ in Figure 8 and NULL exceptions are not collected. This concept of throwing NULL exceptions provides a mechanism of immediately terminating processes in modern programming languages, such as in Java and C++. In case a process performs an illegal instruction, an ordinary exception can be thrown instead of a NULL exception. This exception will be caught by the CSP construct in which the process runs. The CSP construct makes sure that the exception will be collected by the nearest exception construct and a single NULL exception will be thrown further. This way,

duplicated exceptions are avoided and sets of exceptions do not have to be thrown. Furthermore, compatibility is preserved with the try-catch clauses in Java or C++.

The parallel construct will wait until all parallel branches have joined. Subsequently, a NULL exception is passed to the exception construct. See step ❾ in Figure 11. The exception construct catches the NULL exceptions and it will try to heal the registered channel ends. See step ❿. The channel ends of channel *b* cannot be healed and they remain poisoned as long as the channel remains poisoned. After healing, the exception construct will perform process *EH*. *EH* gets the set of exceptions. The set of exception must not be empty, otherwise *EH* can be ignored. The non-empty set of exceptions must be read by *EH*. The set of exceptions does not contain NULL exceptions.

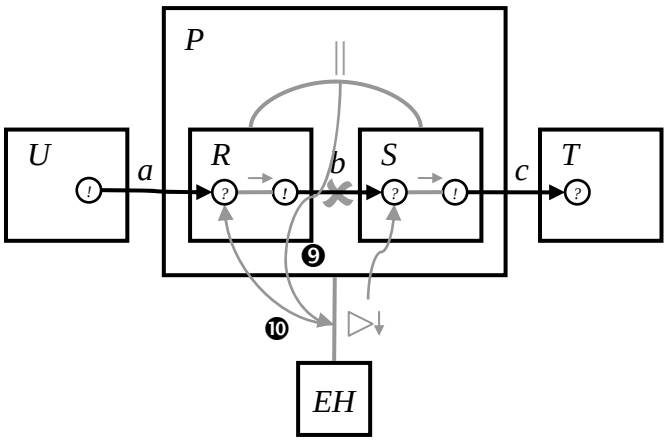


Figure 11. The parallel construct throws a NULL exception, which is caught by the exception construct. The exception construct tries to heal its channel ends before *EH* is executed.

After *EH* has terminated and not all exceptions have been handled, the exception construct will notify the upper exception construct and passes the remaining exceptions to the exception construct in the same way as channel ends do.

This example illustrates that the processes *U* and *T* are not affected by the exception in *P*. In case *U* and *T* must terminate due to an exception in *P*, the program must be designed such that the composition of exception constructs and exception handlers specify this behaviour. The method `refuse()` or `poison()` is not required.

2.3. Example of Nested Exception Constructs

An example of nested exception constructs is illustrated in this section. The steps in the previous described example are also briefly discussed in this example. This example illustrates that an exception constructs can be composed in various ways, which results in nested behaviours. The example will illustrate three kinds of behaviours which can be modelled with this exception handling mechanism. This example is implemented with CTJ.

Figure 12 shows a CSP diagram of the parallel processes *P*, *Q*, *R* and *S*, which model a pipeline of communication via the channels *a*, *b* and *c*. The processes *EHPQ* and *EHR* are in parallel.

The grey arrows in Figure 13 illustrate the registration of channel ends to their exception construct and the registration of lower exception constructs to upper exception constructs. This figure illustrates a complete registration of all elements, i.e. channel ends and nested exception constructs. The same arrows depict the possible paths of notification. The reverse arrows depict the paths of poisoning and healing the registered elements. See Figure 14.

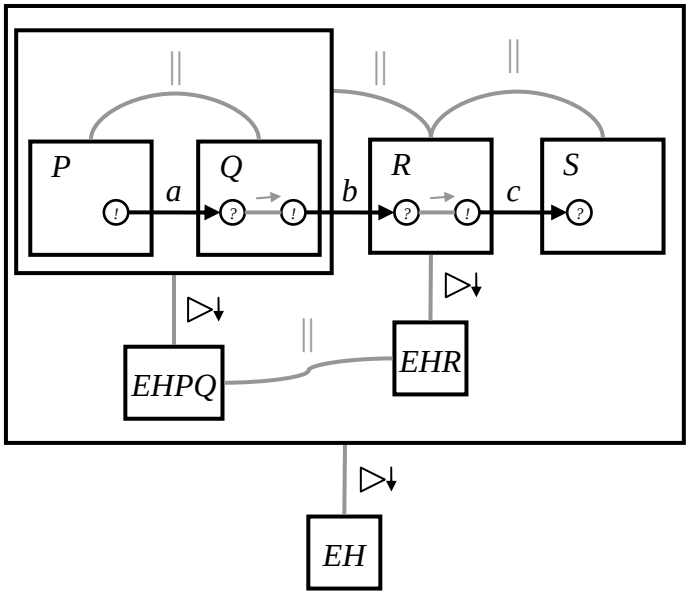


Figure 12. Example of nested exception constructs.

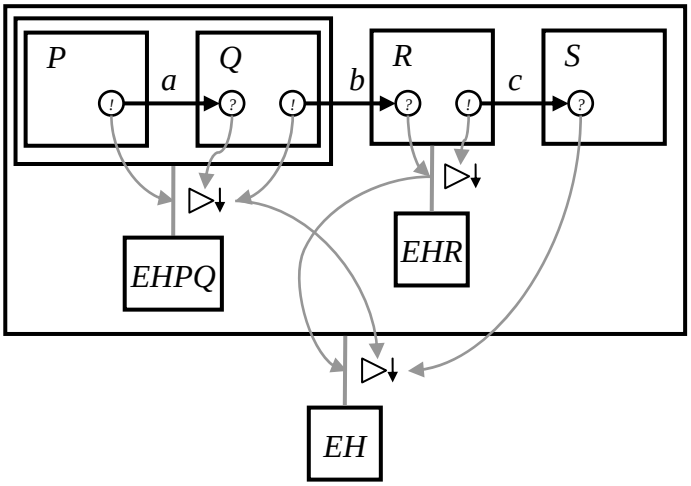


Figure 13. Registering elements to the nested exception constructs.

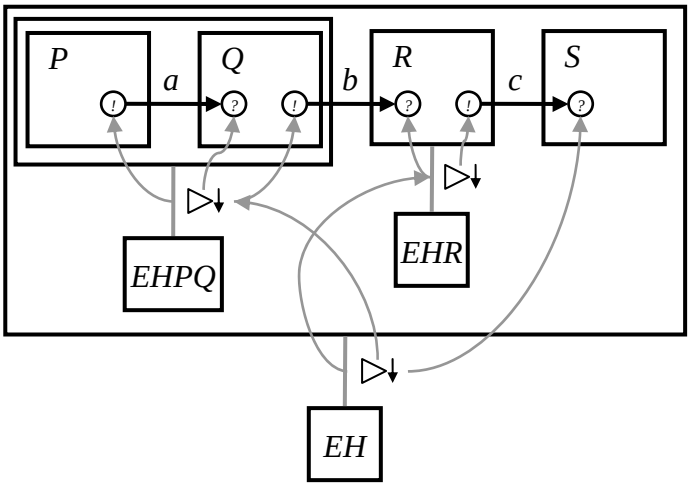


Figure 14. Poisoning or healing elements.

In case channel *a* is in exception and process *Q* is the first process willing to communication via channel *a*, this process is the first to go in exception. That is, process *Q* will stop engaging in any event. The channel end will add the exception to the associated exception construct and throws a NULL exception. This notification starts with the bold arrow between the input of process *Q* and the exception construct. See Figure 15.

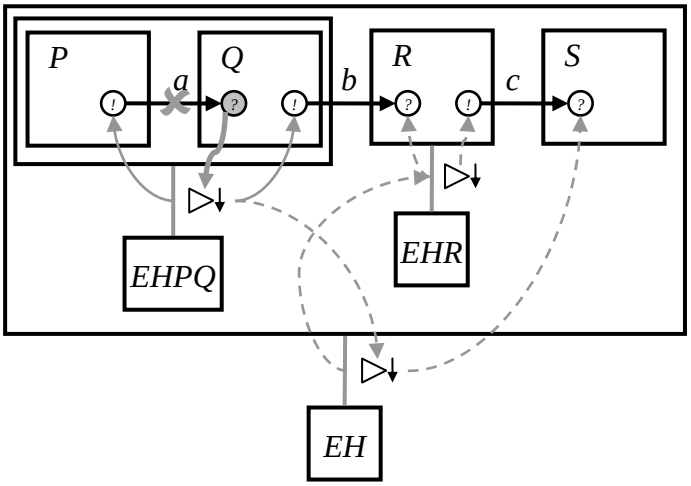


Figure 15. Example of a chain reaction in a nested exception construct.

The exception construct will immediately poison its registered channel ends. The exception remains hidden by the exception construct until the exception handler dealt with the exception and terminates. The exception cannot be observed by the upper exception construct. In case the exception handler terminates and one or more exceptions were not handled, the exceptions become observable by the upper exception construct. The exception will be notified and passed to the upper exception handler *EH*. See the chain reaction of the dotted arrows. The upper exception construct will poison all other registered elements. This makes sure that the sub-processes go into exception. After all sub-processes are in exception and the exception construct catches a NULL exception, the registered channel ends will be healed. Otherwise *EH* cannot reclaim the channel ends. See also Figure 14. After healing, the exception handler *EH* will be executed.

When channel *c* is poisoned then the exception will be added to the exception construct of *EHR* or to the exception construct of *EH*. This choice depends on which thread of control in *R* or *S* was first to execute a channel end of *c*.

Assume process *S* was executed before process *R*. See the bold arrow in Figure 16. This exception starts a chain reaction whereby all process in the scope of the exception construct will be poisoned. An exception construct that is poisoned before it executes will not execute at all. This can happen for the processes *P*, *Q*, and *R* in this example.

In case process *R* outputs on *c* before *S* inputs on *c*, process *EHR* will be executed. If *EHR* uses channel ends then these channel ends will be poisoned when *S* is scheduled and tries to input from *c*. *EHR* will go into exception. However, *EHR* can perform communication events in the meantime. Thus, an exception in channel *c* results in a non-deterministic choice between different traces of events. A trace of events is a sequence of events in which a process can engage. If certain traces of events are unwanted, the following measures can be applied for this example:

1. *EHR* should be designed such that it immediately terminates when an exception occurs on channel *c*, i.e. it must not engage in any communication event. *EH* should take care of the exception, not *EHR*.

2. Process *S* could be executed at a higher priority than *R*, which makes the choice of possible traces of events deterministic.

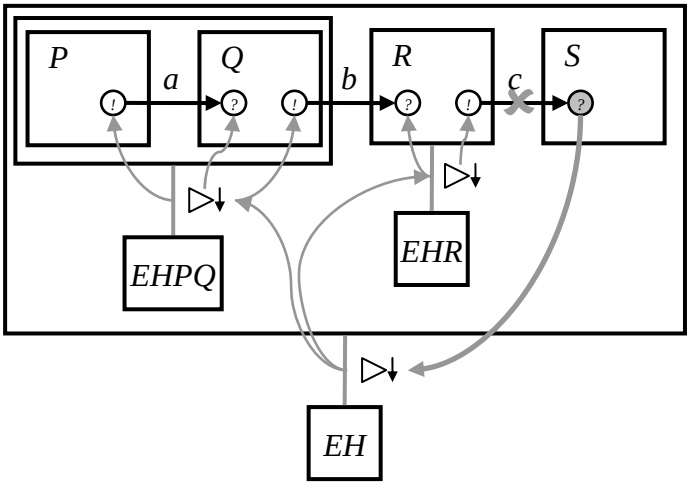


Figure 16. *S* detects exception before *R* on channel *c*.

3. Example program

3.1. Source Code of Program

In this section, the CTJ (Java) code of the example in the previous section is listed. A detailed discussion of the implementation of the exception construct itself is deferred to a later paper.

```
public static void main(String[] args) {

  // Declare the channels and channel ends

  final DataChannel<Integer> a = new DataChannel<Integer>();
  final ChanIn<Integer>  a_in  = a.in();
  final ChanOut<Integer> a_out = a.out();

  final DataChannel<Integer> b = new DataChannel<Integer>();
  final ChanIn<Integer>  b_in  = b.in();
  final ChanOut<Integer> b_out = b.out();

  final DataChannel<Integer> c = new DataChannel<Integer>();
  final ChanIn<Integer>  c_in  = c.in();
  final ChanOut<Integer> c_out = c.out();

  // Declare the processes

  Process p = new Process() {
    public void run()
      throws Exception {
       System.out.println("P: running");
       System.out.println("P: writing to channel a");
       a_out.write(10);
       System.out.println("P: terminated");
    }
  };
```

```
Process q = new Process() {
  public void run()
    throws Exception {
      System.out.println("Q: running");
      System.out.println("Q: reading from channel a");
      int x = a_in.read(null);
      System.out.println("Q: writing to channel b");
      b_out.write(x);
      System.out.println("Q: terminated");
    }
};

Process r = new Process() {
  public void run()
    throws Exception {
      System.out.println("R: running");
      System.out.println("R: reading from channel b");
      int y = b_in.read(null);
      System.out.println("R: writing to channel c");
      c_out.write(y);
      System.out.println("R: terminated");
    }
};

Process s = new Process() {
  public void run()
    throws Exception {
      System.out.println("S: running");
      System.out.println("S: reading from channel c");
      int z = c_in.read(null);
      System.out.println("S: value = " + z);
      System.out.println("S: terminated");
    }
};

// Declare the exception handlers

Process ehpq = new Process() {
  public void run()
    throws Exception {
      System.out.println("EHPQ: running");
      LinkedList<Exception> exclist = ExceptionCatch.getExceptionSet();
      //...
      exclist.removeFirst(); // exception is handled, remove from set
      System.out.println("EHPQ: terminated");
    }
};

Process ehr = new Process() {
  public void run()
    throws Exception {
      System.out.println("EHR: running");
      LinkedList<Exception> exclist = ExceptionCatch.getExceptionSet();
      //...
      exclist.removeFirst(); // exception is handled, remove from set
      System.out.println("EHR: terminated");
    }
};
```

```
    Process eh = new Process() {
      public void run()
        throws Exception {
          System.out.println("EH: running");
          LinkedList<Exception> exclist = ExceptionCatch.getExceptionSet();
          //...
          exclist.removeFirst(); // exception is handled, remove from set
          System.out.println("EH: terminated");
        }
    };

    // Declaring the compositional construct

    Process proc = new ExceptionCatch(
                new Parallel(new Process[] {
                   new ExceptionCatch(
                      new Parallel(new Process[] {p,q}),
                      ehpq),
                   new ExceptionCatch(
                      r,
                      ehr)
                   ,s,
                   }),
                eh);

    // Poison one or more channels to study its effects

    csp.lang.System.refuse(c, new Exception("Exception in channel c"));

    // Start the program

    try {
      proc.run();
    } catch (Exception ex) {
      java.lang.System.out.println("Exception = " + ex);
    }
    java.lang.System.out.println("\nProgram has terminated");
  }
}
```

After a data channel is declared, its input and output channel ends must be received from the channel using respectively the `in()` and `out()` methods on the channel. The processes can read from a input channel end or write on a output channel end.

The references to the channel ends are `final`, which channel ends are allowed to be directly used by the processes. This makes the use of constructors superfluous and keeps the program compact (for the purpose of this paper).

The exception construct is implemented by the process `ExceptionCatch`. An exception handler must retrieve the set of exceptions with

```
LinkedList<Exception> exclist = ExceptionCatch.getExceptionSet();
```

The `getExceptionSet()` method is a read-only static method. The method returns the set of exception. Note: The `ExceptionCatch` plays the role of a call-channel. Any process can invoke the `getExceptionSet()` method. Only exception handlers can retrieve the set of exceptions; otherwise the set will be empty. This also implies that the set of exception can be retrieved by parallel exception handlers associated to the same exception construct.

The exception handler can retrieve the first exception in the set with `exclist.getFirst()` as shown in the example. Since the set is an iteration object, other useful methods are available. After the exception has been handled, it must be removed from the set with `exclist.removeFirst()` or with other methods that are specified by the iteration object. Careful, a race-condition of simultaneously deleting elements must be prevented. Therefore, parallel exception handling must be disjoint. Handling exception twice is asking for trouble anyways.

3.2. Results

In case, channel *c* is poisoned, the three possible paths of abnormal termination are given in the Table 1.

Table 1. Output of the program with channel *c* poisoned.

Result 1	*Result 2*	*Result 3*

`Q: running`
`Q: reading from channel a`
`P: running`
`P: writing to channel a`
`P: terminated`
`Q: writing to channel b`
`R: running`
`R: reading from channel b`
`R: writing to channel c`
`EHR: running`
`EHR: terminated`
`Q: terminated`
`S: running`
`S: reading from channel c`
`EH: running`
`EH: terminated`
 	
`Program has terminated`
	`S: running`
	`S: reading from channel c`
	`EH: running`
	`EH: terminated`
	
	`Program has terminated`
		`Q: running`
		`Q: reading from channel a`
		`P: running`
		`P: writing to channel a`
		`P: terminated`
		`Q: writing to channel b`
		`S: running`
		`S: reading from channel c`
		`EH: running`
		`EH: terminated`
		
		`Program has terminated`

4. Discussion

The steps that are performed by the implementation of the exception construct and channel ends are concurrent paths of executions. These paths of execution must be properly synchronized. This resulted in a multithreaded object-oriented framework that is too detailed for the human mind. Fortunately, the exception construct encapsulates this framework and turns it into a simple and secure design pattern.

The exception handling mechanism has been carefully designed such that the overhead is reasonable low. The overhead is allotted to the process of registering, poisoning and healing of channel ends and nested exception constructs.

A program that does not move channel ends or processes around will register its channel ends and its lower exception constructs only once. For a program that is never in exception this costs an instruction (i.e. a Boolean check which remain false) for each channel end and entering or leaving the exception constructs. In case the program never goes into exception, the exception constructs can be removed from the composition. This design decision lowers the overhead even further. In most cases, there is always one outer exception construct present. For example, this outer exception construct prints the strings of exceptions in the console provided by the operating system.

After the channel ends and nested exception constructs are registered to the upper exception construct, the process of poisoning or healing by the upper exception construct is based on a short list of elements. Poisoning and healing is straightforward, deterministic and light weight.

There can be more than one path of abnormal termination for a single exception. See Section 3.2. The performance of each path of abnormal termination needs to be taken into account for real-time systems. As long as the traces of events are deterministic, the delays will be deterministic. Poisoning channels and processes via the exception construct is faster than gracefully termination [8] and faster than poisoning channel ends by processes [6].

The read and write operations can be viewed as illegal instructions. Hence, throwing exceptions by channels is similar to throwing exceptions by illegal instructions. Therefore this approach does not conflict with the ordinary `try-catch` mechanism in Java or C++.

The application programming interface (API) was not affected by adding the exception construct to CTJ. The protected interfaces of the channel ends required a few additional methods for poisoning and healing the channel ends. These methods are invisible for the user.

A process that performs an infinite loop and which does not invoke channel ends, cannot be poisoned via channel ends. In this circumstance the static method `Expr.evaluate(Boolean expression)` can be used in `while(..)` statements; e.g. `while(Expr.evaluate(i<j)) { … }`. Normally, the method returns the result of the Boolean expression. The surrounding exception construct can poison the method so that it will throw a NULL exception. The loop will immediately terminate.

In future work, the implementation of the exception construct need to be formalized and model-checked in order to prove that the implementation is free from pathological problems, such as race-hazards, deadlock or livelock.

The alternative construct was not discussed in the examples. The alternative construct has been adapted to support asynchronous exceptions. The alternative construct has the simple task not to perform when at least on guard is poisoned. This is obvious, since no legitimate choice can be made when a guard is poisoned. In CTJ, a channel end can play the role of a guard. The exception of each poisoned guard must be notified to the surrounding exception construct, which collects all the exceptions. Subsequently, the alternative construct will throw a NULL exception.

5. Conclusions

We succeeded to build a simple exception construct in CJT for capturing exceptions in concurrent systems. The steps that are required to perform the exception handling mechanism were discussed. This mechanism illustrates the coherency between channel ends, the compositional exception constructs, and the CSP constructs.

Errors in hardware, which cannot be fixed by the underlying software layer, will nicely propagate as exceptions via channel towards the exception handling processes in the program.

The use of exception constructs is orthogonal to the development of processes. Processes that are reused can be included in exception handling without modifying those processes. There is one exception for processes with infinite loops that do not communicate via channels. In this case, a `Expr.evaluate(..)` method is required in the `while(..)` statement, which can be poisoned by the surrounding exception construct.

The detection of an exception at more than one place in a concurrent program defines multiple paths of exception handling. The choice of one of these paths is likely to be non-deterministic due to the non-deterministic behaviour of thread scheduling. This approach

makes non-deterministic paths of exception handling observable and traceable at an appropriate level of abstraction.

The concept of poisoning channels and processes is intuitive and easy to understand. The behaviour of exception handling is attributed to the composition of constructs. This approach is justified in CSP terms. The semantics of this exception construct is informal and need to be formalized in CSP. A full CSP description is in our future work plans. Researchers are invited to contribute.

Acknowledgements

The author wants to thank Peter Welch for his comments and input. Thoughts have been exchanged about formalizing this exception construct in CSP.

References

[1] Hilderink, G. H. (2005). *Managing Complexity of Control Software through Concurrency*, Laboratory of Control Engineering, University of Twente, ISBN 90-365-2204-8.

[2] Hilderink, G. H. and J. F. Broenink (2003). *Sampling and Timing: a Task for the Environmental Proces*, Commmunicating Process Architectures 2003, J. F. Broenink and G. H. Hilderink, IOS Press, University of Twente, Enschede, 7-10 September 2003, Volume 61.

[3] Hoare, C. A. R. (1985). *Communicating Sequential Processes*, Prentice-Hall, London, UK.

[4] Roscoe, A. W. (1998). *The Theory and Practice of Concurrency*, Series in Computer Sciences, C. A. R. Hoare and R. Bird, Prentice-Hall.

[5] Brown, N. C. C. (2004). *C++CSP Networked*, Communicating Process Architectures 2004, I. R. East, J. M. R. Martin, P. H. Welch, D. Duce and M. Green, IOS Press, Oxford Brrokes University, United Kingdom, 5-8 September 2004, Volume 62, pp. 185-200.

[6] Brown, N. C. C. and P. H. Welch (2003). *An Introduction to the Kent C++CSP Library*, Communicating Process Architectures 2003, J. F. Broenink and G. H. Hilderink, IOS Press, Enschede, University of Twente, The Netherlands, 7019 September 2003, Volume 61, pp. 139-156.

[7] Burns, A. and A. Wellings (1990). *Real-Time Systems and their Programming Languages*, International Computer Science Series, Addison-Wesley Publishing Company.

[8] Welch, P. H. (1989). *Graceful Termination -- Graceful Resetting*, Applying Transputer-Based Parallel Machines, Proceedings of OUG 10, Occam User Group, IOS Press, Enschede, Netherlands, April 1989, pp. 310-317.