Cooperative Exceptions for Concurrent Objects

Bruno Cabral, Alcides Fonseca, Paulo Marques
University of Coimbra
Coimbra, Portugal
Email: {bcabral,amaf,pmarques}@dei.uc.pt

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, PA, USA
Email: jonathan.aldrich@cs.cmu.edu

***Abstract*—The advent of multi-core systems set off a race to get concurrent programming to the masses. One of the challenging aspects of this type of system is how to deal with exceptional situations, since it is very difficult to assert the precise state of a concurrent program when an exception arises. In this paper we propose an exception-handling model for concurrent systems. Its main quality attributes are simplicity and expressiveness, allowing programmers to deal with exceptional situations in a concurrent setting in a familiar way. The proposal is centered on a new kind of exception type that defines new paths for exception propagation among concurrent threads of execution. In our model, beyond being able to control where exceptions are raised, the developer can define in which thread, and when during its execution, a particular exception will be handled. The proposed model has been implemented in Scala, and we show its application to the construction of concurrent software.**

***Index Terms*—exception handling; concurrent programming; multi-core; threads; processes**

I. Introduction

Exception Handling (EH) has been around for several decades now. It is by far the most used technique in the development of robust and fault-tolerant software. EH is an essential tool for reliability, since the large majority of modern programming languages rely on EH constructs for dealing with errors and abnormal situations. Unfortunately, dealing with exceptions in concurrent systems poses a challenge significantly different from sequential EH [1]. An EH mechanism should rely on the way the system is structured and be a fundamental part of the system design. But, if we consider the application of traditional EH mechanisms to concurrent systems, some difficulties immediately arise in terms of the concurrent execution of handlers, the signaling of exceptions and the communication between threads or handlers. Although the development of EH models for sequential object-oriented systems has a long history, the same is not true for concurrent object-oriented systems. Research in this area remains very active. Furthermore, most concurrent systems nowadays continue to use sequential EH due to the lack of a definitive model for performing concurrent EH.

The advent of multi-core systems fostered the ”birth” of a new generation of programming languages that will, hopefully, change the way we program parallel systems. These languages include Erlang [2], Fortress [3], X10 [4], and Chapel [5]. Furthermore, the contributions of Concurrent Haskell [6] and Cilk [7] are also of great relevance. However, advances in concurrent programming are not limited to the emergence of new programming languages. Other steps forward have been taken by popular development platforms in order to support new forms of concurrency. Consider, for instance, the examples of the Fork-Join mechanism for Java [8] and Futures [9]. The main objective of these constructs is to help spread concurrent programming by making it much simpler. But, two fundamental issues are frequently neglected or even not addressed at all: how to generally deal with exceptions in parallel systems, and how to communicate and handle abnormal situations in a concurrent environment.

There have been many attempts to create a model for dealing with exceptions in programs where multiple threads of execution cooperate to achieve a common goal or compete for a particular resource. Transactions and transactional systems are a classic example (e.g., [10]). But, other authors have suggested equally valid solutions for performing backward or forward error recovery and for passing exception notifications between threads. Examples of such models include Conversations [11], [12], CA Actions [10], [13], the Guardian [14], and SaGE [15], [16]. These systems are intrinsically complex, and justifiably so because they target systems with strong safety-critical requirements. However, this complexity conflicts with the objective of the recent developments in concurrent programming, which is to simplify such coordination tasks for developers. In many general purpose programming contexts, developers do not require such levels of safety, and a simpler system would serve them better.

Even in the new programming languages mentioned above, exceptions are propagated in the call stack of the thread experiencing problems and ultimately to the parent thread when the execution of the child ends with an unresolved exception. There are very few EH mechanisms that allow the propagation of an exception to a parallel thread without the termination of the thread raising the exception: the mechanisms that do exist require threads to be grouped in order to share information on exceptional occurrences [17] and, consequently, have an inevitable impact in the software design.

In this paper, we propose to eliminate these shortcoming of EH models by allowing the propagation of exceptions among any running thread, independently of hierarchies or forcing threads to end only to raise exceptions. Furthermore, this model enables the separation of concerns, between exception handling code and business code, among distinct threads. Also, to give developers an intuitive and familiar mechanism that builds on constructs they already know, so that they can

```
object Controller extends Actor
  with ExceptionModel {
    def act() {
        val tree =
            TreeFactory.createRandomTree(0);
        val worker = (new Worker).start
        try {
            val i: Any = worker !? tree
            println("Maximum: " + i)
        } catch {
            e: InfiniteValue =>
                println("Infinite value
                present in Tree")
        }
        exit()
    }
}

class Worker extends Actor
  with ExceptionModel {
    def act() {
      loop {
        react {
          case e: Tree => {
            try {
              val ans = e match {
                case n: RealNode => n.value
                case InfNode => {
                  throw(new InfiniteValue)0
                }
                case t: Node => {
                  val r =
                    ((new Worker).start
                    !! t.right)()
                    .asInstanceOf[Int]
                  val l =
                    ((new Worker).start
                    !! t.left)()
                    .asInstanceOf[Int]
                  if (r > l) r else l
                }
              }
              sender ! ans
            } catch {
              e:InfiniteValue => {
                sender ! 0
              }
            }
            exit()
}}}}}
```

Fig. 1. Example of a desired use of exceptions in a concurrent programming

naturally start using it to manage concurrent exceptions in their programs without a steep learning curve, our model integrates seamlessly with existing `try-catch` type EH mechanisms in modern programming languages. However, the formal description of the programming language constructs introduced with this new model, due to its size and high relevance, is out of the scope of this paper and is depicted in a separate work.

Our main contributions are:

- The definition of a new EH model supporting the propagation of exceptions among any number of threads executing in parallel, without imposing unwanted design constraints, and without causing termination of the thread raising the exception;
- The implementation of the proposed model as an extension to the Scala [18] language.
- The release of the complete source code and several programs illustrating how the model can be used to effectively handle exceptional conditions in concurrent software.

The outline of this paper is as follows: Section 2 discusses motivation, background and related work; Section 3 introduces the EH model being proposed; implementation details are depicted in Section 4; in Section 5 we present the results from a preliminary validation of the model; and in Section 6 we lay the conclusions.

II. Background and Related Work

In this section we discuss the background that influenced this work and was fundamental to determining the requirements for our approach to concurrent EH. Please note that, when we speak of participants or members in a parallel computation, or processes or threads of a concurrent program, we are referring to the same thing - any kind of thread of execution that runs in parallel within the same software application.

A. Motivation

The main problem with modern concurrent programming languages, in terms of EH, is their inability to communicate exceptions between threads executing in parallel. Consider, for instance, the example in Figure 1. This is an example of a parallel program, based on the Actor model, which looks for the maximum value among the nodes of a tree. In the event that an infinite value is found in one of the nodes, the program should stop, since it is no longer possible to identify the maximum value. The `Controller` creates a tree with a random number of nodes and values in each node. The `Controller` also launches a `Worker` to initiate the search from the top of the tree. The `Worker` checks if the value on the node under analysis holds a real or an infinite value and, in the case it is a real value, it compares it with the maximum computed from the analysis of the branches of the node. The relevant part for our work is what happens when the node holds an infinite value. In this situation, the `Worker` should be able to notify all the other `Workers` and the `Controller` executing in parallel that execution should stop. A natural way to handle this problem is to use the EH mechanism. EH were specially created to handle such abnormal situations. Unfortunately, the ”exceptional” code in the example would not produce the desired effect. Current EH models do not allow exceptions to be propagated to concurrent threads unless the thread raising the exception is hierarchically related with the thread receiving the exception notification and the former has already been terminated. Thus, in this example the actor executing in parallel would remain completely unaware of the occurring exception. Our approach, described in this article, removes this limitation.

B. Exception Handling

Exception handling is a civilized way of dealing with exceptional situations (occurrence of a condition that changes the normal flow of execution of a program.) Exception-like language constructs have been around since the mid-1950s [19]. In the 1960s, facilities for dealing with exceptional conditions, such as variable overflow, end-of-file, and bad data, were fairly common in programming languages. But, it was not until the development of the IBM PL/I programming language [20], [21] that we saw the usage of high level control flow constructs exclusively dedicated to enabling the writing of reliable and safe programs. In 1975, John B. Goodenough [22] suggested a notation for EH and, later, Flaviu Cristian [23] defined its usage. Since then, we can argue that programming language constructs for handling and recovering from exceptions have not changed much despite being present in all mainstream programming languages.

Unfortunately, the execution semantics for such mechanisms are best suited for implementing sequential programs. When we try to apply them in parallel applications, we face some challenging issues:

- Neither static propagation [24], [25] or dynamic propagation models (which search the call stack to locate an adequate handler) designed for sequential applications allow the propagation of exceptions among two or more distinct threads executing in parallel. This is true even if those threads are hierarchically related (e.g., as in the Java threading model);
- The search for a suitable handler is either performed on the lexically related code or by checking the code of enclosing stack frames. Thus, it is impossible to execute exception handling code on a distinct thread, i.e., other than the one raising the exception, when using the EH mechanism alone;
- Control flow, both after the occurrence of an exception or after the execution of a handler, is only defined for sequential code;
- Raising an exception on a running thread does not have, per se, any direct influence on the execution of a concurrent thread, even when they have strong interdependencies;

C. Support for EH in recent concurrent programming languages

Despite the just mentioned pitfalls, sequential EH remains as the most used form of EH in parallel applications to the day. Looking at how EH is used in, or extended by, recent concurrent programming languages and libraries, we can point out some important shortcomings on the way exceptions are handled.

In some languages, such as Chapel [5], the EH model is not well understood. But, even in concurrent languages where the EH semantics are well known, things are not much better. In Scala [18], designers opted to keep the same exception model of Java, in which, as we know, the native EH mechanism does not directly support communicating exceptions between threads. On the other hand, the Actor model available for Scala allows an actor to monitor another actor and check if it ends with an exception or in a regular fashion [18]. Other languages, such as Fortress, X10, JCilk, and Erlang allow a process, upon its termination, to communicate its termination status to the immediately hierarchically superior process (or thread). In the case of an abnormal termination, the exceptional information is also communicated [3], [4], [17], [26], [27]. Nonetheless, the remaining EH semantics on these languages are similar to the ones present in traditional models. Erlang is a small exception to this rule: it offers an extra EH feature where processes can be linked together in a way that guarantees that whenever one process crashes, all associated processes receive a notification, and can either die or trap the notification signal [17]. Despite representing an advance in the way exceptions are handled on concurrent systems, the fact is that a computation still has to terminate in order to communicate an exception occurrence. Thus, any recovery attempt will always require another thread to execute the recovery action for the failed computation. Furthermore, in terms of arrivals, exception notifications are completely asynchronous. The receiving process can, in some cases, be more affected by the interruption caused by the notification arrival then by the exceptional occurrence itself. Managing groups of processes that share notifications among them can also be cumbersome if there is a heterogeneous set of exceptions and/or diverse relationships between processes and exceptions.

Taking a different perspective, the introduction of new mechanisms for concurrency into existent platforms, such as Futures and Fork-Join, also means that these platforms require more functionality in terms of EH. E.g., [28] identifies several difficulties encountered when dealing with exceptions in such systems.

As the discussion above illustrates, despite the vast and recent advances in concurrent programming, EH continues to be poorly supported in parallel programming models. As the authors in [29] concluded, designers of programming languages often neglect the exception mechanism and look at it more like an add-on for their language instead of a central part. As a consequence, software quality inevitably suffers.

D. Design for concurrency

The detection of an abnormal situation in the execution of a parallel program presents different challenges depending of the computation at hand, and requires the intervention of a varying number of the participants in that computation. Consider that:

- If an exception cannot be handled inside the failing participant it should be communicated to other participants, who are capable of (1) dealing with the exception, by themselves or cooperatively, and allowing the failing component to recover, or of (2) replacing the failing participant, or of (3) preventing their own execution from continuing.
- In any case, if the exception can affect the overall computation or a subset of the participants, the failing component has to be able to communicate the exception

to whoever might be affected. Furthermore, the exception handlers on all the involved parts must also be able to communicate the result of their recovery actions among themselves.
- Communication of exceptions between all the participants of a computation, or between their exception handlers, must be done without forcing the premature termination of any member.

These needs cannot be fulfilled by current `try-catch` EH models. Other approaches, specific for handling exceptions in concurrent and distributed applications, can indeed provide such functionality but, as we will discuss further in the Related Work, they might be difficult to use and intrusive, and developers are not eager to use exceptions for recovery purposes [30].

E. Related Work

The list of references in terms of related work is vast. But, even so, there is no definitive and widely accepted solution for dealing with concurrent EH.

Concurrent EH systems can be broadly divided into two major groups: transactional and non-transactional. In terms of transactional error handling systems, probably the most successful is the Coordinated Atomic Actions (CA Actions) scheme [13]. CA Actions combine and extend the concepts of atomic actions [11] and Conversations [11], [12]. The execution of a CA Action looks like an atomic transaction for the outside. Inside of a transaction, participants cooperate and interact through local objects. In the presence of an exception each participant is forced to handle it independently of the fact of which participant first observed the abnormal occurrence.

The CA Actions scheme has its limitations. External exceptions are explicitly signaled from a CA Action participant thus, in some cases, it might not be possible to detect abnormal conditions outside the participants. Furthermore, concurrently signaled exceptions are expected to be related in some way so that the exception resolution mechanism can pull off a meaningful result. But, the main problem with concurrent exception handling is determining which is the correct handler to invoke (it may be different for each participant) Relying on handler communication to ensure the correct handlers are invoked may be a highly complex task- The authors of the Guardian model [14] have addressed this issue. The Guardian model, contrary to what happens in the prior model, does not raise the same exception in all action participants to notify the occurrence of an abnormal event. By raising in each process a possibly different exception and specifying the context in which it should be handled by the process, the Guardian model guides each process to a correct exception handler, thus orchestrating the recovery action. No transaction-like structure is needed for the correct exception handlers to be invoked (though that structure may be useful for other reasons). [14] Thus, we can fit the Guardian model into the non-transactional EH mechanisms category.

The SaGE system [15], [16], as happen in the Guardian model, is targeted to Multi-Agent Systems (MAS). This framework proposes a layered handling approach depending on Java call stack structure which has been extended for differentiating higher level exceptions from language level exceptions. According to the authors, SaGE does not rely on the use of entities external to agents and integrates exception handling mechanisms in the execution model of the agents. SaGE supports cooperative concurrency and manages the propagation of exceptions between cooperating agents by, for instance, transforming exceptions into more representative ones in terms of the whole system.

All these approaches, in a certain way, require developers to define groups of threads, processes or agents in which only some types of exceptions are recognized and can be dealt with. In many situations, these systems also introduce boundaries between the execution of groups of process, establishing frontiers for the propagation of exceptions, distinguishing types of exceptions, and defining intricate rules for handler selection. The same is true in more recent approaches, such as Failboxes [31]. Our model, on the other hand, does not introduces so many concepts or imposes such design restrictions.

In terms of programming languages, Erlang [2], [17] also supports some form of grouping of processes. A group of process can be linked together in order to allow all the process in the group to be notified when some exception occurs in a worker process. Erlang distinguishes manager processes from worker process. Typically an Erlang program is a tree of supervisor processes whose leaf nodes are workers. A downside of this approach is that the worker commits suicide after communicating the exception to the upper manager processes and, commonly, these will do the same.

In languages such as Java [32], C++ [33], or C# [34], there is no form of support for concurrent EH. They do not even allow exceptions to be propagated outside the thread where they were raised. Thus, many of the most recent concurrent programming languages represent a real step forward in terms of EH. Most of them are better equipped to handle concurrent exceptions. But, improvements are not as profound as it is desirable. Fortress [3], X10 [4], JCilk [27], and Scala [18] simply allow a failing thread/process to communicate an exception to its parent when it terminates its execution in an abnormal way.

Akka [35] is an actor library and microkernel for Java and Scala, inspired by the Reactive Manifesto [1], which uses the concept of the "Supervising Actor" for dealing with abnormal situations. The actor is responsible for handling its childrens failures, forming a chain of responsibility, all the way to the top. When an actor crashes, its parent can either restart or stop it, or escalate the failure up the hierarchy of actors. The Akka EH model does not allow propagating an exception among non-hierarchically related actors.

DOOCE [36] allows multiple exceptions to be thrown concurrently in the same `try` block to be handled concurrently. Also, a `try` block can have multiple `catch` statements. In DOOCE, exception handlers begin their execution after all

[1]The Reactive Manifesto, Version 1.1 (http://www.reactivemanifesto.org), September 23, 2013.

subtasks in the protected region terminate, either normally or abruptly, or if any of the participated objects throws an exception. In our model, the execution of the handlers, in each thread, for a particular exceptional occurrence can be initiated immediately after the arrival of the exception notification. In addition to the termination model [32], DOOCE also supports resumption [22].

Arche [37], [38] proposes two kinds of exceptions: global and concerted. Global exceptions are thrown and communicated to other process when a process terminates abnormally. Concerted exceptions are the result of the execution of a customized resolution function that takes all exceptions thrown concurrently as input parameters. Concerted exceptions are prepared in order to allow them to be handled in the context of the calling object.

The as-if-serial EH [28] model allows asynchronous exceptions to be propagated to the invocation point of the function call of a Future as if the call is executed locally, whereas in the prior systems, exceptions are propagated to the thread that spawns the computation when it attempts to synchronize with the spawned thread.

Software Transactional Memory (STM) sets the ground for making protected blocks atomic[39]. Operations inside a `try-catch` are either all completed or none is. And if there is an exceptional occurrence, there is a `catch` clause that is able to restore the system. Although the behavior is what we expect, STM solutions today are still very expensive. For instance, the authors of the Atomic Boxes mechanism [40] report slowdowns between 10 and 1000 times for simple programs featuring sorting algorithms such as *quicksort* and *bubblesort*. Furthermore, if the exception is caused by external factors (writing to a file and other IO operations), the system cannot automatically recover from those, which also poses a problem.

III. Cooperative Exceptions for Concurrent Objects: A Proposal

In this section, we propose a new EH model for dealing with exceptions in concurrent systems that aims to give developers an intrinsic ability to communicate and handle exceptions cooperatively among parallel execution threads and, at the same time, retain the same economy of expression present in sequential EH models.

A. Approach

The Cooperative Exceptions for Concurrent Objects (CECO) model introduces a new kind of exception, with different semantics from existing checked and unchecked exception models [32], [41] or from the EH models in modern concurrent programming languages [18], [2], [3], [4], [5], [17], [26].

Exceptions in CECO are represented by objects that can, not only be propagated on the call stack of a thread, but also between parallel threads of execution. While the exception is propagated between threads and also along the call stack, the information about its path is recorded in the exception object itself.

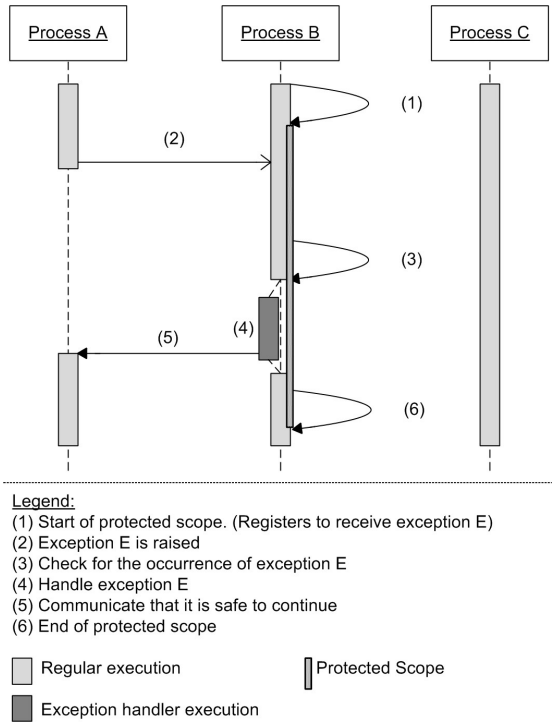

Fig. 2. Exception propagation and handling

Figure 2 illustrates the raising, propagation and handling of an exception using CECO. The graphic represents three parallel threads of execution, in this case named Processes A, B, and C. Independently of the type of computation at hand, the scenario that we are describing shows how an exception raised inside one of the processes can affect the execution of other processes in the same application. Consider that the three processes have been executing in parallel for some time and performing whatever tasks they were programmed to do. At some point during its execution, Process B decides that it should be alerted if one of its peers raises an exception of type E. To express such a wish, Process B bounds a section of its code (1), thus establishing the scope in which it expects to be notified of the occurrence of any exception E that might be raised by the processes executing in parallel. Later, during its execution, Process A detects an abnormal situation and decides to raise exception E. Since B registered to receive this type of exception, the system propagates the exception to B (2). Note that Process A and C do not get the exception, C never having registered for receiving it and A, despite raising the exception, also never having registered to receive it. In the example, the protection scope in process B ends at (6) and B will no longer be notified of the occurrence of exception E. A process will only receive notification of exceptions that are delivered to it after it has begun executing inside the protected scope; it is impossible for a process to be informed of what exceptions occurred prior to that moment. Moreover, if the execution of

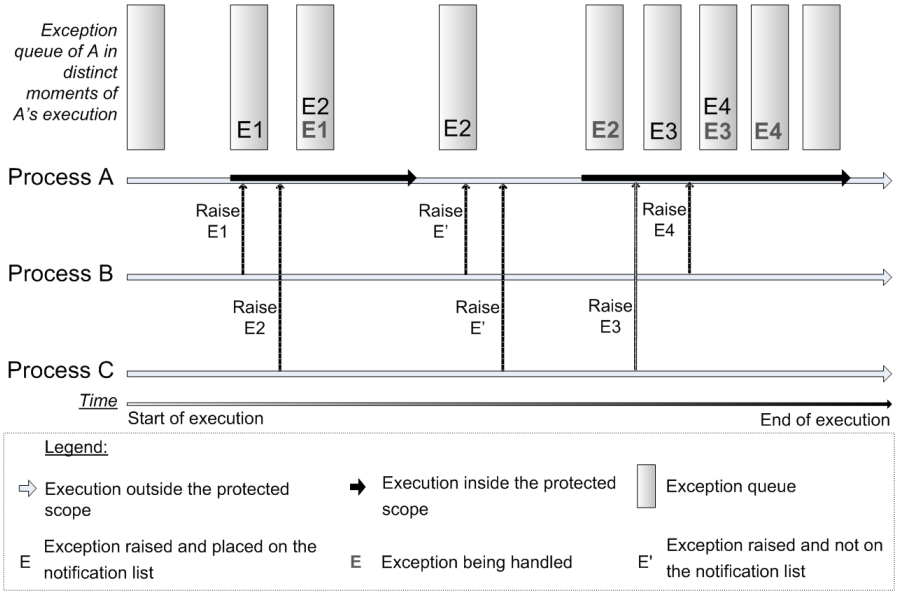

Fig. 3. Arrival and notification of concurrent exceptions accordingly to a process protected scope

a process moves away from the protected region of code, any concurrent exceptions that might be delivered afterwards will no longer be delivered and are discarded for that process alone. Protected scopes are defined by `try-catch` blocks.

Contrary to what happens in the Checked Exceptions model, a concurrent exception does not need to be forcefully handled by other threads. In our approach, a thread may decide to ignore some exceptions and deal with others. If a thread does decide to acknowledge the occurrence of an exception and does not deal with it, the exception will be propagated up the call stack of that thread, and eventually cause the thread's termination.

Another key aspect of CECO is that, even after being notified of the exceptional event, the execution of Process B may not be immediately interrupted. Developers may opt to explicitly indicate the location where the program should check for the occurrence of some types of exceptions (3). This is what we call *Explicit Reception* of an exception. This feature is most useful in parallel applications in which processes need to be aware of problems with cooperative processes, but in which an unexpected interruption would be more harmful than not knowing about the exception. The alternative, which we also provide, is an *Implicit Reception* mechanism that ensures that an exception is delivery asynchronously any time after a process is registered for receiving it. This usually means than the occurrence of such an exception in a concurrent process can interrupt the execution of the current process, even if it is only momentarily.

When Process B effectively receives exception E, the handler for that exception inside B is activated. Process B will then conduct the necessary operations for making the system safe for A and/or B (4). In this particular scenario, if B is successful, it communicates to A that it is safe to continue (5). Afterwards, A and B can continue with their regular execution if recovery was successful. If A had not stopped its execution and had just continued executing, B would not have to send any notification and could just continue on. Stopping process A was a design option for this example and was not derived from any kind of restriction on the model. The communication sent from B to A is an example of a customized recovery strategy; it is not a part of CECO, and it could be implemented using whatever mechanism the developer prefers.

Figure 3 represents the execution of a process that is part of an application composed by 3 concurrent processes. During its execution, process A alternates between sections of the code where it enables the reception of concurrent exceptions, and others where it does not. Consider the case where during its execution, inside the protected scope, the process receives notification of 2 exceptions, E1 and E2, and starts to handle the first one. If the handling of this exception causes the process to abandon the protected scope (e.g., by throwing a local exception), E2 will remain in the notification list. Thus, if the process falls again under a protected region where it can handle E2, the exception is delivered at that time. On the other hand, if other exceptions E' are raised while the execution is between protected regions, they will not be communicated to the process. Thus, from the series of exceptions raised during the process execution – *E1 E2 E' E' E3 E4* – only E1, E2, E3, and E4 are communicated to the process, since they were raised while the process was executing inside a protected region. Note that the process registers to receive particular types of concurrent exceptions; it is not required to receive all types of concurrent exceptions. Thus, if a concurrent exception of an unexpected type is raised while the process is executing inside a protected region, it will not be delivered to that particular process. CECO guarantees that a process receiving

two exceptions from another process gets the notifications in the same order the exceptions were raised, and that no duplication occurs. There is no guaranty that exceptions raised by different processes are globally delivered in the same order, as that would require global synchronization. This is a design issue left up to implementations; if programmers are willing to pay the scalability cost of global synchronization, then stronger guarantees on ordering can be provided. In any case, exceptions will be delivered to a process one by one, with only one handler activated at a time.

B. Syntax and Semantics

CECO introduces a revised set of programming language constructs and new types for representing exceptions compatible with all mainstream object-oriented languages.

From this point on, we will refer to `CException` (short form of *Concurrent Exception*) as the type of the exception objects that can be communicated between different processes. `CException` objects contain information about the kind of abnormal situation that was encountered, a time-stamp and point of origin (object, method, thread of the occurrence.) We will also refer to the EH models used in languages such as Java, C#, C++, Scala, and others, as "tradional EH models" or "sequential models".

To propagate `CException` objects between processes, the process that encounters an abnormal situation must perform one of the following instructions:

```
raise exception
      or
throw exception
```

Where `exception` is an object of type `CException`. The `raise` instruction makes the `exception` occurrence known to any concurrent process that is currently registered for receiving this type of exception (`try-catch` for a CException). The process raising the exception continues its execution at the next lexical instruction immediately after the `raise` command.

On the other hand, the `throw` instruction not only propagates the exception to any parallel process but also throws the exception on the raising thread. `throw` has the same effect as the traditional `throw` (as the ones found in Java), but also propagates the exception to concurrent processes. In this situation the execution in the throwing thread continues into the next suitable `catch` handler. It should also be noted that calling `raise` can also trigger an exception handler in the raising thread if the process is executing inside a `try-catch` block with a specific handler for that type of exception (we will discuss the semantics of `try-catch` next.)

The argument of the `raise` command has to be of the type `CException` or a descendant. But, if `throw` is invoked with a non-`CException` parameter it means that the exception is local and is to be propagated in that process as it is in sequential EH models.

With the `throw` and `raise` commands, processes are able to raise concurrent exceptions in an asynchronous fashion. But, these notifications will not have any effect on other parallel processes if none is registered for receiving such type of exception. For registering to receive a certain kind of concurrent exceptions, processes can resort to a language construct similar to the traditional `try-catch` blocks.

```
try {...} catch (<concurrent exception>) {...}
                    or
async_try {...} catch (<concurrent exception>) {...}
```

The semantics of these blocks are very similar to their traditional counterparts, but if a `catch` references an exception of the type `CException`, the process is in fact registering for receiving concurrent exceptions of that kind for the duration of the execution of the complete `try-catch` block. This is a fundamental difference between CECO's `try-catch` and the traditional one, the protection scope for concurrent exceptions comprises not only the `try` section, as it is common, but also the `catch` blocks when they are executed. This means that any concurrent exceptions being raised during the execution of a `try-catch` block will be handled by the handlers of that block, even if they are raised while the code in the `catch` part is executing. This means that another handler can be activated immediately after the termination of an handling action. Nonetheless, the execution of these handlers is dependent on the nature of the `try-catch` block, which can be synchronous and asynchronous, as we will see next. `try-catch` blocks can also be nested inside other `try-catch` blocks. And, `try-catch` blocks can have `finally` blocks similar to the traditional blocks with the same syntax.

The `try-catch` blocks allow concurrent exception notifications to be delivered either *synchronously* or *asynchronously*:

- *Synchronously* - This is the default delivery scheme for concurrent exceptions in CECO's `try-catch` blocks. This means that even if a process registers itself to receive a certain kind of exception, and even if such exception occurs during the execution of the respective `try-catch`, it will only be effectively perceived by the recipient process when it reaches a *safe point* in its execution. These *safe points* are identified by the keyword `check`, which we will discuss next. Nonetheless, the sequential semantics still apply to local exceptions. Thus, if a function is invoked inside the `try-catch` block and throws a local exception, execution is immediately aborted and transferred to the next suitable handler if it exists.
- *Asynchronously* - In this scenario (`async_try`), the developer might consider it to be more practical to interrupt the execution of the receiving processes as quickly as possible without having to wait for a *safe point*. After handling the exception, execution at the receiving thread continues on the first instruction after the handler, or in another handler if different exception arrived.

When a process is executing inside the protected scope for concurrent exceptions, that is to say inside a `try-catch` block where it has registered for receiving such type of exceptions, and the default synchronous delivery policy is in effect,

the process needs to reach a *safe point* in order to acknowledge any pending notification. Safe points are identified with the following instruction:

```
check
```

The execution of `check` is only allowed inside `try-catch` blocks. A `check` will verify if any of the "expected" concurrent exceptions was raised during the execution of the protected scope. If one or more of such exceptions were raised, the first one (the oldest) is raised locally upon returning from the `check` operation. Execution is then transferred for the most suitable local handler. If there are no pending notifications, execution continues normally after the `check` operation. Exceptions raised by the `check` operation are eliminated from the queue of pending notifications of that particular process, but will continue to exist in the notification queues of other processes.

IV. Implementation

We have chosen Scala [18] for implementing and testing of CECO. Scala was selected because it is a modern programming language, which has been attracting considerable attention from the academia and the industry, and that clearly targets the development of concurrent applications. Scala has the particularity of allowing us to implement a language extension in a very simple way, similar to what we would do to create a common library. This is possible because Scala allows masking some special methods as if they were language keywords. Also, Scala has inherited Java's EH model and, consequently, inherited all the limitations of this model for dealing with abnormal concurrent events. Thus, one might argue that the implementation of CECO for Scala represents a useful complement to the language. The CECO's source code and several coding examples are available for download[2]. Scala Actors (inspired on the Actor Model) can inherit structure and functionality from the CECO language extension.

To avoid undesired inconsistencies with the existent exception handling constructs in Scala, we decided to add "_" as a prefix to CECO language constructs.

A. Framework

CECO language constructs can be seamlessly integrated with Scala code and used for implementing concurrent exception handling strategies within the Actor Model.

1) Implementation details: Actors should be extended with the `ExceptionModel` trait in order to inherit the features required for using the concurrent EH model. Concurrent exceptions can be triggered by both the `_throw` and the `_raise` commands. Under the hood, when either one of these two methods is invoked, the concurrent exception being raised is sent to a global actor named `ExceptionController`. This actor is responsible for managing the communication and delivery of exceptions among actors. The `ExceptionController` accepts registrations from actors for the reception of exceptions. And, since every actor extending `ExceptionModel` has its own notification queue for concurrent exceptions, `ExceptionController` is also responsible for dispatching concurrent exceptions to exception queues of the registered actors.

For registering to receive concurrent exceptions, actors must use either one of the following combinations: "`_try{...}_catch {...}`" or "`_tryF{...}_catch{...} _finally{...}`", or any of its asynchronous counterparts "`_async_try{...}_catch {...}`" or "`_async_tryF{...}_catch{...}_finally{...}`". The type of exception expected on the `_catch` block defines the type of concurrent exception that the Actor will be monitoring. Between entering the `_try` block and leaving the `_catch` block, the actor is registered for receiving exceptions of the `E1ConcurrentException` type.

Note that, if during the execution of `_try{...}_catch {...}` more than one concurrent exception is communicated to the actor, the `_catch` handler will be executed at least once for each one of them in order of reception. If an exception of a local type is raised, forcing execution to leave the boundaries of the current block, the reception of the concurrent exception ceases until a new block is initiated. In case of nested blocks, reception will continue active until the outermost block is terminated.

Invoking `_check` inside a `_try{...}_catch {...}` block causes the top exception, on the local pending concurrent exceptions queue, to be dequeued and thrown locally using the native Java/Scala exception mechanism.

Internally, `_async_try{...}` are executed as regular `_try{...}_catch {...}` blocks, but in a different thread. This thread does not run on the regular actor thread-pool because we need to keep its reference for future interruption. If a concurrent exception must be notified to that block of code during its execution, we use Java/Scala's `InterruptionException` mechanism to interrupt its execution and jump to the adequate handler.

B. Coding examples

The following examples will help to understand how CECO works in practice and include a traditional *Master-Worker* and a *Pipe-And-Filter* design. These examples, and others, are available on-line [3].

1) Batch Mailing System: This program is composed by two kinds of Actors: The *Controller* and the *Senders*.

The Controller possesses a list of e-mail addresses to which it must send e-mail messages. To perform this task, the Controller spawns multiple new Sender actors. Each Sender is responsible for sending a message to one address.

Senders can raise two types of concurrent exceptions: `IOException` and `InvalidAddressException`. When any of these is raised, the Sender will exit immediately. This behavior is promoted by the use of the `_throw` method.

[2]https://github.com/AEminium/ceco

[3]https://github.com/AEminium/ceco/tree/master/src/main/scala/examples

```
...
while (!recipients.isEmpty) {
  senders = List()
  _try {
    recipients.zipWithIndex.foreach { m =>
     val ms = new MessageSender
      ms.start
      ms ! (m)
     senders = ms :: senders
    }
    recipients = List()
    senders.foreach{ m => m !? Stop }
    _check
  } _catch {
    e:ConcurrentException =>
      e match {
        case e:
         InvalidAddressException => {
          database = database - e.email
          println("[Controller] " +
             e.email
          + " does not exist")
      }
      case e:IOException => {
        recipients = e.email :: recipients
        println("[Controller] IO error")
}}}
...
```

Fig. 4. Main loop of the Controller.

The Controller is registered to receive notification of both types of exceptions. But, it behaves differently in each case:

- When it receives an `InvalidAddressException`, the Controller removes the address from the database.
- When it receives a `IOException`, the Controller retries to send the message to the same address using a second (or third) Sender.

The Controller will only check for exceptions after sending a full batch, guaranteeing that all addresses are tried at roughly the same time.

Figure 4 shows the main loop of execution of the Controller.

To implement this example without CECO, one simple approach would be to modify the main actor to receive exceptions as messages. Whenever a worker actor finds an exception, it has to send that exception as a message back to the main actor. The control flow for the main actor has to cope with the fact that exceptions are retrieved from its inbox only after executing the main loop. This has the disadvantage that exceptions cannot be caught in the middle of the processing of one message, only in between. Another major criticism for this approach is mixing both normal and exceptional information on the same channels. One of the major accomplishments of all EH mechanisms is to provide the means to impose the separation of concerns between normal and exceptional execution.

2) Pipe-And-Filter: The second example mimics the production line of a factory. There are four units in the line: a Receiver, a Preprocessor, a Processor and finally a Dispatcher unit. Each unit has a queue of elements to process, and can only process one element at a time. Elements are moved from one unit to the next in a Pipe-And-Filter way. The Receiver puts new elements into the line and Dispatcher handles their

```
def work(u:Int) = {
  _async_try {
    println("[2] Preprocessing " + u)
    //the actual work
    Thread.sleep(3*1000)
    println("[2] Preprocessed " + u)
    next ! u
  } _catch {
    e:ProcessingFault => {
      println("[2] Reason to stop: " + e.i)
      discardAbove = e.i
      _raise(new PreprocessingFault(e.i))
    }
  }
}
```

Fig. 5. Work function of the Preprocessor.

finalization. Preprocessor and Processor units are responsible for handling the elements.

The elements being processed by this production line have to be obligatorily handled in their arrival order (FIFO). Since all the units in the line can be processing different elements in parallel, if one unit fails, the work on the preceding ones has to stop. But, the units in front can continue to function normally. Furthermore, the elements on the failing unit and in the prior ones have to be re-processed in the initial order and from the beginning. The purpose of this example is to demonstrate the usage of the asynchronous `_async_try` blocks. Each unit main work cycle is protected by one of these blocks, thus work can be immediately interrupted in case of a failure in a proceeding unit.

In the actual implementation of this example, the Receiver gets 5 elements to process. However during the Processing phase of the third element, a machine jam is simulated. When it happens, the previous area in the pipeline, Preprocessor, is notified, discards the current element as well as all the ones in the queue. Finally, it raises a `PreprocessorEx` meaning that it cannot work (due to problems in the later machine). The Receiver unit is notified and restarts delivering elements again from 3, maintaining the expected order. Figure 5) presents the main work function of the Preprocessor unit and the usage of the `_async_try` blocks.

To imitate the functionality of this example, developers would have to write a library mimicking almost all the functionality of the CECO model. Developers would also have to mix a thread-based approach with the Actor model. Without CECO as a language extension, it would be impossible to perform asynchronous delivery of exceptions without breaking the Actor model. And, one could also argue that exceptional and normal execution paths would not be differentiated as it is desirable for exception handling.

V. Preliminary Validation

To conduct a preliminary validation of the usage of CECO in the development of real world software, we decided experiment with two common industry scenarios: a) the creation of a program from scratch; and b) the adaptation of an existent application to incorporate CECO. We asked a professional developer, with more than 10 years of experience and no previous

knowledge of CECO, to design and implement/upgrade these programs.

A. Desktop File Search

The first case study, the Desktop File Search software, was implemented from scratch. Actually, two versions of the program were created: one using CECO and another without CECO. This allowed to compare the effort, complexity and solutions involved in the creation of both versions. The Desktop File Search program allows users to perform file searches over a Linux file-system, and was implemented from scratch.

The program is structured using the Actor model and was designed to promote modularity. The FileCrawler and the FileScanner actors are responsible for finding and identifying all accessible files in the system. The Indexer actors builds in-memory indexes of existent files but, have limited memory space. Thus, when a index is full, it cannot reference more files, and new index actor has to be created. In the CECO implementation, when the indexer in unable to handle new requests it raises a `NewIndexerException` which is caught in the Monitor actor. The Monitor is then able to create a new Indexer. The Searcher actor inquires the indexers in order to find possible all possible paths of a file. In the version not using CECO, there is no Monitor, the Indexer itself, when confronted with an `ArrayIndexOutOfBoundsException` sends a message to the FileScanner object to notify him that it cannot index more files and the FileScanner actor creates a new Indexer.

B. Scalairc bot

The second case study focused on improving the exception handling policies in the Scalairc[4] bot program, an IRC Client written in Scala, using CECO.

The ConnectionChecker actor notifies the Control Actor whenever it needs a Connection re-initiated.The Control actor uses the Connection actor to initialize a connection. Unfortunately, the Control actor is not aware of the occurrence of `java.net.SocketTimeoutConnection` and `java.net.UnknowHostException` raised in the Connection actor, thus the Connection actor can fail to create a new connection and the Control actor has no immediate way of acknowledging that abnormal behavior.

C. Results

The first case study showed us that there was no significant difference in the number of lines of code effectively written in both versions of the program: a) 305 sloc in the CECO version; and b) 348 sloc in the regular version. The developer learned how to use CECO by studying the example programs and by listening our explanations. Unexpectedly, learning to code with CECO took less than half of the 60 minutes that we had initially planned for training the developer. The similarity, in terms of code, with the traditional EH models was essential in this process. In terms of effort, the developer reported having spent more than 90 minutes and less than 120 minutes to code each version. But, CECO had the advantage of keeping exceptional code and communications totally separated from business code. The traditional approach forced the developer to communicate exceptional events through ”regular” channels and increased the complexity of message handling code. In terms of performance, after an initial battery of runs of both versions of the program, we did not notice any noticeable difference in execution times, either in presence or absence of exceptional situations. Thus, we decided not to pursuit this line of validation further.

On the second case study, the Connection actor raised concurrent exceptions using CECO, which were handled in the Control actor immediately. With this knowledge, the Control actor could try a different strategy for making another (existent or new) connection available to requesting actors or, in case a severe problem is detected, forward an exception to other actors. Without CECO, the developer would need to create a supervisor actor to facilitate the handling of exceptions. But, it would not be trivial and very intrusive to mingle exception handling code with business code, which is why it is understandable why the programmers decided to ignore the problem in the first place, instead of dealing with it. CECO helped to make the application more reliable without the extra baggage. The modifications implied almost no change in the original code and the extra coding was actually far less then we predicted. Of course, the amount of extra code is directly dependent on the complexity of the exception handling actions.

VI. Conclusions

In this paper we propose a new model for dealing with exceptions in concurrent software applications. Our objective is to give developers a system that they will effectively use. The examples on this article show that our approach is straightforward, efficient and highly adaptable to any concurrent architecture. Our model provides a well defined propagation model that communicates concurrent exceptions only in relevant paths, prevents unexpected interruptions of the processes/threads trapping these concurrent exceptions, and allows concurrent exception handlers to cooperate harmoniously in combined recovery actions. Moreover, all this is possible without resorting to any form of complex process/thread grouping, or having to terminate threads only to communicate the occurrence of an exception. Also, by allowing unrelated processes to exchange exception information among them directly only by raising an exception (which is impossible or not trivial to do in other models), we can quickly prevent programs from continuing executing on an unfeasible path.

Future works include a formal description of the model, a new implementation for Java, and a larger case study. Furthermore, there is room for evolving CECO, thus, we are convinced that the synchronization between handlers must become a concern of the EH model, in order to assist in the development of more powerful cooperative exception handling strategies.

[4]https://github.com/colder/scalaircbot

REFERENCES

[1] A. Fonseca and B.Cabral, "Handling Exceptions in Programs with Hidden Concurrency: New Challenges for Old Solutions", in ICSE'12 Workshop Proceedings, 5th International Workshop on Exception Handling (WEH.12), 2012.

[2] J. Armstrong, R. Virding, C.Wikstrom, and M.Williams, "Concurrent programming in Erlang", Prentice Hall, Second Edition, 1996.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele Jr.,S. Tobin-Hochstadt, "The Fortress Language Specication - Version 1.0", Sun Microsystems, March, 2008.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing", in Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05), ACM, New York, NY, USA, 2005

[5] B. L. Chamberlain, D. Callahan, H. P. Zima, "Parallel Programmability and the Chapel Language", in International Journal of High Performance Computing Applications, Vol.21(3) August, 2007.

[6] S. Peyton, A. Gordon, and S. Finne, "Concurrent Haskell", in Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL '96), St Petersburg Beach, Florida, ACM, 1996.

[7] C. F. Joerg, "The Cilk System for Parallel Multithreaded Computing", PhD thesis, MIT, 1996.

[8] D. Lea, "A Java fork/join framework", in Proceedings of the ACM 2000 Conference on Java Grande (JAVA '00)", ACM, San Francisco, California, United States, June 03 - 04, 2000.

[9] C. Flanagan, and M. Felleisen, "The semantics of future and its use in program optimization", in Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05), ACM, San Francisco, California, United States, January 23 - 25, 1995.

[10] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu, Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery, in Proceedings of the 25th FTCS - International Symposium on Fault-Tolerant Computing, Pasadena, USA, 1995.

[11] R. H. Campbell, B. Randell, Error Recovery in Asynchronous Systems, in IEEE Transactions on Software Engineering, Vol. SE-12(8), 1986.

[12] B. Randell, A. Romanovsky, C. M. F. Rubira-Calsavara, R. J. Stroud, Z. Wu, and J. Xu, From Recovery Blocks to Concurrent Atomic Actions, in Predictably Dependable Computing Systems, ESPRIT Basic Research Series, Springer-Verlag, Brussels, 1995.

[13] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, A. F. Zorzo, D. Schwier, and F. von Henke, "Coordinated Atomic Actions: Formal Model, Case Study and System Implementation", Technical Report, UMI Order Number: NEWCASTLE-CS#TR98-628, 1998.

[14] R. Miller, A. Tripathi, The guardian model for exception handling in distributed systems, in Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS 2002), IEEE, Piscataway NJ, USA, 2002.

[15] F. Souchon, C. Dony, C. Urtado, S. Vauttier, "A proposition for exception handling in multi-agent systems", in Proceedings of the SELMAS03 International Worskshop, 2003.

[16] F. Souchon, S. Vauttier, C. Urtado, C. Dony, "Fiabilit des applications multi-agents: le systme de gestion dexception sage", 2004.

[17] J. Armstrong, "Making Reliable Distributed Systems in the presence of Software Errors", PhD thesis, 2003.

[18] M.Odersky, L.Spoon, and B.Venners, "Programming in Scala - A comprehensive step-by-step guide", First Edition, Artima, November, 2008.

[19] J. McCarthy, P. W. Abrams, D. J. Edwards, T. P. Hart, and M. Levin, LISP 1.5 Programmers Manual, MIT Press, 1965.

[20] International Business Machines (IBM), IBM System/360 PL/I Reference Manual, SRL Form C28-8201-1, 1968.

[21] G. Radin, The early history and characteristics of PL/I, in History of Programming Languages, Academic Press, 1981.

[22] J. B. Goodenough, Exception handling: issues and a proposed notation, In Communications of the ACM, Vol. 18(12), ACM Press, December 1975.

[23] F. Cristian, Exception Handling and Software Fault Tolerance, In Proceedings of FTCS-25, 3, IEEE, 1996 (reprinted from FTCS-IO 1980, 97-103).

[24] J. L. Knudsen, Exception handling - a static approach, in Software: Practice and Experience, Vol. 14(5), May 1984.

[25] J. L. Knudsen, Better exception handling in block structured systems, in IEEE Software, Vol. 4(3), May 1987.

[26] J. Danaher, "The jcilk-1 runtime system", Masters thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, June 2005.

[27] I. Lee, "The JCilk multithreaded language", Masters thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, August 2005.

[28] L. Zhang, C. Krintz, and P. Nagpurkar, "Supporting exception handling for futures in Java", in Proceedings of the 5th international symposium on Principles and practice of programming in Java (PPPJ '07), ACM, New York, NY, USA, 2007.

[29] A. Garcia, C. Rubira, A. Romanovsky, and J. Xu, A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software, in The Journal of Systems and Software, Vol. 59(2), 2001.

[30] B. Cabral and P. Marques, Exception Handling: A Field Study in Java and .NET, in Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP 07), LNCS 4609, Springer-Verlag, 2007.

[31] B. Jacobs, F. Piessens, "Failboxes: Provably safe exception handling", in proceeding of: ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009.

[32] J. Gosling, B. Joy, G. Steele, and G. Bracha, The Java(TM) Language Specification, 3rd Edition, Prentice Hall , June 2005.

[33] B. Stroustrup, "The C++ Programming Language", third edition, Addison-Wesley Longman Publishing Co., Inc, 2000.

[34] ISO/IEC, Information Technology Programming Languages - C#, 2nd Edition, ISO/IEC, Switzerland, 2006.

[35] Akka Scala Documentation - Release 2.4-SNAPSHOT, Typesafe Inc, June 5, 2014.

[36] S. Tazuneki, and T. Yoshida, "Concurrent exception handling in a distributed object-oriented computing environment", in International Conference on Parallel and Distributed Systems: Workshops,IEEE Computer Society, Washington, DC, USA, 2000.

[37] V. Issarny, "An exception handling mechanism for parallel object-oriented programming: Towards reusable, robust distributed software", in Journal of Object-Oriented Programming, Vol.6(6), 1993.

[38] V. Issarny, "An exception handling model for parallel programming and its verification", in Conference on Software for critical systems, 1991.

[39] C. Fetzer and P. Felber, Improving program correctness with atomic exception handling, Journal of Universal Computer Science, vol. 13, no. 8, pp. 10471072, 2007.

[40] D. Harmanci, V. Gramoli, and P. Felber. 2011. "Atomic boxes: coordinated exception handling with transactional memory", in Proceedings of the 25th European conference on Object-oriented programming (ECOOP'11), Mira Mezini (Ed.). Springer-Verlag, Berlin, Heidelberg, 634-657, 2011.

[41] ISO/IEC, Information Technology Common Language Infrastructure (CLI) Partition I to VI, 2nd Edition, ISO/IEC, Switzerland, 2006.