Exception Handling with Resumption: Design and Implementation in Java

Alexander Gruler
Dept. of Computer Structures, University of Ulm†
89069 Ulm, Germany

Christian Heinlein
Dept. of Computer Structures, University of Ulm
89069 Ulm, Germany

ABSTRACT— Nowadays, nearly every mainstram programming language includes an exception handling mechanism. Basically, the variety of these mechanisms can be devided in two groups according to their handling models: Mechanisms which support exception termination and those which support resumption. Even though resumption was still an attractive alternative to termination for the first exception handling mechanisms in the 1980s, it has nearly been displaced completely in contemporary languages by termination. Some reasons for this trend will be reviewed in this work. But in our opinion, they do not justify to reject resumption completely. To the contrary, resumption can simplify some tasks extremely while still being implementable with reasonable effort. To show this, we discuss some basic concepts of how a resumption mechanism should be designed in general and propose new syntactical constructs to support these concepts. Based on these concepts, we describe the implementation of a prototypical precompiler for Java supporting exception handling with resumption. In fact, the mechanism implemented in the precompiler not only extends, but completely subsumes the Java mechanism, for it is capable of both, termination and resumption.

KEYWORDS— Flexible Exception Handling, Java, Resumption, Prototypical Implementation, Precompiler.

I. Introduction

WITHIN the last two decades of software evolution, offering an exception handling mechanism (EHM) has become a key feature in modern programming language design. Almost all modern EHMs of various programming languages separate the "normal" code of a program from its error–handling code in order to meet the increasing modularity demands of programs and to provide a comfortable way of dealing with errors. However, this separation necessitates to think about the control flow of a program in connection with an EHM.

At present, in most mainstream languages such as Java or C++, the common way of directing the control flow after a responsible exception handler has been found and executed, is *termination*, i.e. *not* returning to the point where the exception has occurred and resuming the program's execution from there, but instead unwinding the (process) stack and continuing execution at the level of the respective handler. We agree with B. Meyer, the designer of Eiffel, that the only reasonable alternative to termination is *resumption* with the motivation to "cure" the error that has caused the exception and to continue/resume the original execution afterwards [1].

This work describes, how the EHM of Java and other languages can be extended by the ability to resume execution after completion of the respective handler, at the point where the exception has been raised. Beforehand, in the remainder of this section, we define a terminology and recall the advantages and disadvantages of resumption. In section II we will have a look at existing EHMs that support resumption. In section III, we will clarify some general, conceptual issues concerning the design of an EHM that supports resumption: How should such a mechanism in general be designed to provide a safe and (for the programmer) comprehensible way of dealing with exceptions and to overcome the disadvantages of former EHMs. Of course, these resulting concepts are applicable to any resumption mechanism, i.e. independent of a specific programming language. Based on these general concepts, we describe in section IV the construction of a precompiler for Java that extends the existing EHM of the language. We have deliberately chosen Java as the "target language" for the precompiler for several reasons: First and foremost, Java already offers a sophisticated EHM that provides a solid basis to build onto. Also, some lingustic properties of Java have turned out to be very suitable for the implementation in that they allow to keep it on a very high level of the language, i.e. to avoid descending into the very fundamental mechanisms of a compiler (e.g. changing the program counter, stack pointer, etc. explicitly). Furthermore, Java is a very popular, modern language which guarantees that the concepts will reach a large variety of people. Section V concludes the paper.

A. Terminology

For the sake of clarity, this section briefly introduces a terminology that will be used throughout this work. A system that executes programs has to be able to deal with unexpected events that only occur during runtime and that are usually not predictable at compile time. Such an unexpected event or abnormal condition is commonly referred to as an *exception*. Systems that support signalling, handling and detection of exceptions are said to provide *exception handling* (EH). The *exception handling mechanism* (EHM) of a system determines how exceptions affect the control flow of a program.

During the execution of a program the system is always in one of two states: *normal activity* or *exceptional activity*. The occurence of an exception makes a system change from normal to exceptional activity. This means, that the *standard continuation* of the program is replaced by

† Author's new address: Chair IV, Software & Systems Engineering, Technical University of Munich, 85748 Garching, Germany

an *exceptional continuation* [2], i. e., the program will continue differently after an exception has occurred.

An exception is caused either by an error or by any other event that occurs during the normal execution of a program, preventing the program from following its normal *execution path* [3] and leading it into an *exceptional state*. The method (procedure, routine) or the block of code in which the exception occurs is referred to as the *faulting method* or the *faulting block*.

The specification of a *protected region* allows to define handlers for a certain block of statements. A *handler* is the place where control is transferred to, after an exception has been raised. Since a handler usually constitues a scope and/or a block of its own, we also speak of the *handling context*. Usually different handlers can be specified (for the same protected region) to react on different kinds of exceptions. The *handling model* of an EHM determines how/where to continue after the completion of the responsible handler. In a handling model that supports resumption, we call a handler that continues (or that might continue) execution at the raising context a *resuming handler*.

The signalling of an exception is called *raising* or *throwing*. The place where an exception is raised will be referred to as the *raising point*. The current environment (block, method) containing the raising point is called the *raising context*. After an exception has been raised, it is propagated (automatically or explicitly) through the handler hierarchy until an appropriate handler is found.

B. Arguing about Resumption

Since the first EHMs have been included in languages like CLU [4] or PL/I in the mid 1970s , language designers have increasingly tended to offer termination solely. Reasons for that are often given as: the utility of resumption does not justify its "cost," or resumption makes the control flow of a program too difficult to understand and therefore more error-prone. The developers of CLU (which was beside PL/I the first language that included an EHM) rejected resumption, *"because it was complex and also because we* [the developers of CLU] *believed that most of the time, termination was what was wanted. Furthermore, if resumption were wanted, it could be simulated by passing a procedure as an argument ..."* [4].

Actually, Bjarne Stroustrup presented similar reasons why he has chosen termination as the only handling model for C++ [5]. Firstly, he based his decision on the experience reports of several language designers which have been using EHMs for a long time, while constructing and/or working with languages like for example Cedar and Mesa. Some of these designers even were early proponents of resumption, but have obviously changed their minds, because during the years, the use of resumption in their systems decreased gradually and has always been replaced by *"a more appropriate design"* [6]. Stroustrup presents some other reasons to reject resumption in [7]: The fact, that a part of a program might gain control back due to resumption, is a problem to the extent, that the handler might have changed the context of the faulting block/procedure. So, after the completion of the handler, essential conditions or assumptions about the state of the environment at the raising point might now be different compared to how they were before the execution of the responsible handler. Therefore, he points out that, with resumption, *"raising an exception ceases to be a reliable way of escaping from a context"* [7]. In general, he doubts conditional resumption and suggests to simulate it by a function call instead, for in his opinion, resumption is only a "[...]*non-obvious, and error-prone form of co-routines.*" [7].

Certainly, the reasons given by Stroustrup are not wrong and we agree with him to the extent, that the nature of resuming exception handlers is very similar to the one of function calls. Actually, the implementation proposed in section IV makes use of this property by internally transforming resuming handlers into local methods. But as it will also turn out in section IV, this does not do the job completely: Assuming that we would mimick resumption with methods only, what should the mimicking method do for example, if it cannot cure the cause for the exception? In this case, usual termination (i.e., unwinding the stack) is what is actually wanted. In such a situation it would make no sense at all to resume, which means for the method to return. This is only one example why a language should support resumption with built-in, linguistic facilities and not leave the whole job to the programmer.

The problem, that a handler might change the environment of the raising point, can also be solved, if we allow a resuming handler to "return" some information back to the raising point and in that way to "inform" the raising context about the changes made during its execution. This is analogous to a signalling statement which also transports information in form of the thrown exception (object).

With these and similar minor changes, the only point of criticism that remains, is that resumption is rarely used and therefore the effort to implement it is always too high. But in our opinion, this reason alone does not justify to reject resumption completely, in particular if the effort of implementing such a mechanism remains moderate, as we will see in section IV. Besides, there are indeed many realistic situations where the use of a resumption mechanism can simplify things significantly. E.g., while an OutOfMemory Error is nearly always treated as final and not curable, it could be solved in certain scenarios by freeing data structures (in a handler) which were kept in the memory only for performance reasons, and resuming execution at the raising context afterwards.

Another example is the processing performed by a parser: During the syntactic analysis a parser reads and analyses the grammatical structure of a given input, with respect to a given grammar. If the input does not obey this grammar, it signals an error, usually by raising a kind of "Parse Exception". Therefore, with an EHM based on termination, a very simple parser can be constructed by simply raising an exception on encountering a grammatical error in the input. Of course, such a parser is not capable of detecting

more than one error in the input in a single run for the EHM always terminates after the first error. But nearly every contemporary parser is able to detect and to report multiple errors in the input file in a single run over the input. Such a parser usually uses a special technique: On encountering an error in the input it tries to "repair" the error by guessing an alternative to the currently parsed (erroneous) token and adjusting the input accordingly. Hoping that the guess was right, the remaining input can then be parsed. This technique could be implemented very comfortably using a resumption-mechanism: On encountering an error, the parser simply throws a corresponding exception. The responsible handler adjusts the input according to the guess and resumes afterwards. The parser can now proceed to check the remaining input for errors.

II. Related Work

Of course, resumption is not a new concept for a handling model and, as seen in the last section, since the introduction of the first EHM in languages in the mid 1970s, language developers have been arguing and thinking about resumption. Resumption was already offered by earlier languages like e.g. PL/I, Mesa, BETA, and Lore, but "newer" implementations of a resumption mechanism are rarely found: R. Govindarajan presented a work about EH in functional programming languages [8], where he clearly separates between termination and resumption mechanisms. Schreiner (et al.) describes how resumption can be simulated in regular Java [9]. His article is the basis of our implementation, but as we will see in section IV, it is by no means sufficient for a complete implementation of a resumption mechanism for a programming language. But beside this, for popular, contemporary langagues, there is no known implementation of a resumption mechanism, in particular not for Java and C++.

The EHM offered by Eiffel [10] actually does not provide resumption, but only a retry mechanism, which is a combination of resumption and termination, even though, strictly speaking, *retry* is only a special form of termination. The fundamental difference between resumption and retry is, that a retry mechanism unwinds the process stack exactly like with termination and re-executes the whole faulting procedure a second time instead of continuing execution directly after the raise statement, as resumption does. Thus, in Eiffel, a handler can be attached to a whole routine by adding a `rescue` clause to the routine's definition which may contain a `retry` statement which causes the system to unwind the stack and to re-execute the whole routine a second time.

A signalling construct (`throw-accept`) is also offered in the Lisp [11] language family. But here, it is not intended for EH, but to provide a more drastic form of a `return` to exit from a block and transfer control back through several function calls. However, it does not offer resumption.
More interestingly is the `condition` mechanism of Lisp, for it offers a rudimentary kind of resumption. Conditions in Lisp are what exceptions are in other languages. They can be signalled by many different operators. One of them is the `check-type` macro, that allows to signal a *correctable error* [11]. This means, that the handler for the condition enables the user to provide a corrected value.

BETA [12] offers a more complete resumption mechanism based on its pattern mechanism which replaces types, classes, functions, and also exceptions. In particular, a *virtual pattern* dealing with an exception is called an exception [12]. Exceptions can be associated with class or procedure patterns. The default action for an occurence of an exception (`Exception` pattern) is the termination of the whole program, but this may be avoided by explicitly specifying `Continue` within the description of an exception pattern. Resumption (also called "partial recovery") can be achieved by defining *labels* and restart the execution at such labels from within a handler by means of a `restart` operator. However, BETA does not allow to pass information along with a `restart` (which is a major conceptual disadvantage; see section III).

III. Conceptual Design

Most of the points of criticism, that were presented so far, can be avoided by designing the resumption mechanism appropriately. Therefore, we discuss in this section some design issues of a resumption mechansim. As a result, we propose a general syntax, that overcomes the points of criticism as well as some disadvantages of other EHMs supporting resumption.

A. The `resume` Statement

The first question that arises, is how or, in particular, *where* to indicate resumption. Basically, there are only two possibilities: Firstly, the decision whether to resume or not can be made at the raising point, i.e. by the raise statement itself. This implies that a language would have to offer two different raise statements: one for the termination model and another one for resumption, i.e. where the handler always "returns" and resumes execution at the raising point. The main advantage of this possibility is, that there is no doubt about the continuation of the control flow. In particular, it is already known in the raising context, whether a handler will resume or not.
But is this feasible? Usually only after having tried to cure the cause of an exception, we can say, whether the attempt was successful or not. Therefore, only the handler of an exception can decide, whether it could cure the cause for an exception or not. And, as already emphasized in section I, this knowledge is essential, because resumption only makes sense with the motivation to cure the cause for the exception before resuming normal execution. Therefore, we suggest, that the respective handler should indicate, whether to terminate or to resume.
This indication is best done by means of a new `resume` statement which might optionally appear within a handler. Consequently, the missing of such a statement (or the fact that it is not executed at runtime) indicates termination. A `resume` in a handler is similar to a `return` in a function:

Both end the execution of the respective environment. Like a `return` in a function, a `resume` can basically occur everywhere in a handler, i.e. not only at the end.

The motivation behind resumption was, that the cause for an exception has to be fixed by the responsible handler, before the program can continue normally. Therefore, a resuming handler usually has to perform some changes in the state of the system. But the raising context of the respective exception still assumes the old state of the system, as it was at the time when the exception has been thrown. This means, that a resuming handler needs a facility to inform the handling context about the changes that were made. From another point of view: this information itself can contain and/or could be seen as the *solution* for the cause of the exception.
Passing information from the handling context back to the raising point is best done by allowing the `resume` statement to have a "parameter" analogous to the parameter of a `throw` statement in Java. Thus the `resume` statement has the following form, where *<expression>* denotes the "parameter" which is evaluated and passed back to the raising context:

```
<resume_stmt> ::= "resume" <expression> ";"
```

In languages where exceptions are represented as objects it is sufficient for the `resume` statement to have a single paramter, for the parameter itself can be an object of a certain type (and have multiple fields). As we shall see in section III-B, such a parameter can be accessed in the raising context.

B. `accept` Blocks

We recall that one point of criticism against a resumption mechanism was, that the state of the handling context might have been changed during the execution of the respective handler. Therefore, after the handler has resumed, the code following the raise statement cannot assume that the state of the environment (states of objects, variables, etc.) is still the same as before the exception was raised.
Therefore, the code following the raise statement has to be able to deal with different situations, depending on what changes the respective handler has made before resuming. The design of the EHM should support the programmer to deal with these different situations. For this purpose, we introduce so-called `accept` blocks which can be attached to a `throw` statement like e.g. `catch` handlers to a `try` block in Java. In particular, the specification of an `accept` block also contains the declaration of a variable of a certain type. An EBNF definition of the syntax is shown below:

```
<throw_stmt> ::= "throw" <expression>
                 (";" | (<accept_block>)*)
<accept_block> ::= "accept"
                 "(" <type> <identifier> ")"
                 "{"  <block>  "}"
```

So, every `accept` block contains instructions to deal with a single kind of situation in the raising context. But, how does the EHM determine which `accept` block has to be executed? This is exactly the point, where the `resume` statement and `accept` blocks fit together: The type of the object returned by the `resume` statement determines the `accept` block that will be executed, in the same way as the type of the object thrown by a `throw` statement determines the `catch` block that will be executed. So, depending on what kind of solution the handler has performed to cure the cause for the exception, it sets the parameter of the `resume` statement appropriately to tell the raising context which changes it has made.
So, in a resumption mechanism, we basically need two symmetric facilities: Firstly (as with termination), a raise statement together with respective handlers for signalling an exception (in the direction from the raising point to the handler) and secondly, a `resume` statement in combination with one or more `accept` blocks (for sending the solution back from a resuming handler to the raising context.).

Figure 1 shows a small example (using a Java-like syntax): Method `a()` calls method `b()` within a protected region (`try` block), which has, beside another handler, a handler for an exception of type `Exception2`. The handler contains a `resume` statement that "resumes" (returns) an object of the user-defined type `Solution`. Somewhere within method `b()` an exception of type `Exception2` will be thrown and consequently be caught in the respective handler in method `a()`. In this scenario, the variable `error_is_curable` evaluates to `true` and therefore the handler resumes and returns an object of type `Solution`. Thus, execution continues with the `accept` block for the type `Solution` in method `b()`. When the end of the `accept` block has been reached, the system continues normally with the first statement following the `throw-accept` construct, i.e. following the *last* `accept` block.

```
public void a() {
    try {   b();   }
    catch (Exception1 e1) { ..... }
    catch (Exception2 e2) {
      /* Try to cure the cause. */
      if (error_is_curable)
        resume new Solution("the solution");
      else { /*Clean up and proceed*
              *as with termination.*/ } }

public void b () throws Exception2 {
    ......
    throw new Exception2("Caused by error")
    accept (Solution s1) { ..... }
    accept (AnotherSolution s2) { ..... }
    ..... }
```

Fig. 1. A simple resumption scenario demonstrating the new syntax.

IV. Implementation

This section presents a *prototypical* implementation of a resumption mechanism for Java, i.e. an implemenation that does not aim to be optimized in terms of performance or reduced EH overhead [3]. However, it shows that a

resumption mechanism is rather easily and at the same time affordably to construct. Since a complete description of the implementation is far beyond the scope of this paper, we can only outline the basic ideas and principles — however, a complete description of the implementation is given in [13]. In order to demonstrate the concepts proposed in section III, we have decided to integrate the resumption mechanism into regular Java by constructing a precompiler that transforms the new constructs necessary for the resumption functionality into regular Java constructs. The result of this transformation can be compiled and executed by every Java Virtual Machine (JVM).

In particular, we wanted to provide an implementation that is 100% compatible to existing Java code which does not know anything about a resumption mechanism and only uses the regular EHM of Java. The price for this downward–compatibility were some changes to the conceptual design which will be described in sect. IV-A.

The implementation itself was done at a "high level" of the language, i.e. only the existing EHM of Java and some other language constructs are used. So we rely exclusively on what is offered by standard Java — in particular, the implementation does not set the program counter or the stack pointer explicitly, nor is it necessary to dig deeply into the runtime environment of the system itself by means of the Java Debugger Interface (JDI) or anything alike. Considering the implementation effort, the fact that the implemenation can be done at a high level disproves especially the point of criticism that implementing a resumption mechanism has to be very expensive.

A. Necessary Adjustments to the Syntax

According to the conceptual design of the resumption mechanism, it was *not* necessary to have different kinds of handlers for termination and resumption, respectively. But in order to be compatible to existing Java code, we have to introduce a new kind of handler, the `recover` block.

Imagine the situation, where for example a method `x()` includes a protected region with a *resuming* handler for exceptions of type T. Within this protected region, method `x()` calls another method `y()`, which was developed prior to the introduction of the resumption mechanism (e.g. a library routine). This method `y()` knows nothing about a resumption mechanism, but nevertheless could use the regular EHM of Java. If `y()` now coincidentally throws a regular (terminating) exception of type `T`, the attached handler for `T` in method `x()` would be responsible and would handle this exception. But as soon as the handler tries to resume, the system would crash, because the EHM tries to resume execution at the raising point, which is the method `y()`. But method `y()` certainly never was designed to deal with a resumption, i.e. it only includes a regular `throw` statement without any `accept` blocks.

To avoid such errors with existing code, an implementation has to provide separate handlers for the resumption and the termination model respectively. We emphasize again, that this is only necessary to stay compatible with existing Java code, that uses the regular EHM of Java!

Therefore, the old `catch` block remains the handler for terminating exceptions only. For terminating *and/or* resuming exceptions, we introduce a new construct: a `recover` block/handler. Depending on whether a `recover` block contains and excutes a `resume` statement or not, it continues with resumption or termination, respectively. Syntactically, a `recover` handler looks exactly like a corresponding `catch` handler in Java, except for the keyword `recover`. However, the handlers are completely independent: A `catch` block handles only terminating exceptions, i.e. exceptions that were thrown by a simple `throw` statement, while a `recover` block only handles exceptions thrown by a `throw-accept` statement. In particular, a `catch` and a `recover` handler for the same type do *not* interfere with each other.

B. Code Transformation

The precompiler itself can be constructed quite easily with JavaCC [14]. Actually, the bigger challenge is how to transform the new statements for resumption into corresponding, *regular* Java code. This section presents these transformations and the resulting output of the precompiler.

1) Basic Ideas: In a paper on EH in C++ [7], B. Stroustrup suggests that resumption could be achieved through ordinary function calls. Schreiner has concretized this idea in another article on exceptions [9] and proposes a basic skeleton of how this can be achieved in Java. The implementation described here largely bases on his ideas.

So, in order to implement a resumption mechanism without actually setting the stack pointer excplicitly (as with `goto` constructs or the `setjmp`/`longjmp` facility of C++), we have to simulate the part of exceptional activity in which the user might possibly resume, with method calls. Since, in this implementation, only `recover` handlers can possibly resume, every control transfer to a `recover` handler has to be transformed by the precompiler into a method call. Hence, to achieve resumption, a `resume` statement in a `recover` handler can be directly mapped into a `return` statement of the implementing method. If the `recover` handler does not contain or does not execute a `resume` statement , we use the original EHM of Java to unwind the process stack by throwing a (real) exception and catching it at the appropriate place (the `try` block defining the original `recover` handler) and thus achieve/simulate termination. Normal `catch` handlers are not touched by the precompiler and therefore will work in the usual manner. Actually, all original constructs for EHM of Java will not be transformed at all (except for original `throw`s, `return`s, `break`s or the keyword `this` *within* `recover` blocks).

Since `try` blocks and thus also `recover` handlers can be defined nestedly, a kind of propagation mechanism has to be provided to ensure that exceptions thrown with a `throw-accept` statement are processed/caught by the appropriate `recover` handler. This is done by constructing a *handler stack* which is implemented as

a linked list that represents the calling hierarchy of the `recover` handlers during runtime. Every element of this stack is a throwable object derived from the class `TryRecoverBlock` [1] , that represents a `try` block with at least one `recover` handler attached to it. The objects will be created, appended and removed from the stack dynamically during runtime whenever an affected `try` block will be entered or left.

2) Transformations in Detail: With the possibility of local classes, Java offers an ideal facility to transform `recover` blocks into real method calls and to construct the handler stack. For every `try` block possessing at least one `recover` handler, the precompiler inserts the definition of a local class `MyTryRecoverBlockNN` ("`NN`" in the class name is a unique integer number) directly before the respective `try` block and basically copies the original code of the body of the `recover` blocks into the public method `raise()` which is defined within this local class. `MyTryRecoverBlockNN` itself extends the superclass `TryRecoverBlock`. After the definition of the inner class, an object of it, `selfNN`, is created. Here, the "`NN`" in the object's name is the same integer value as in the corresponding local class's name. (This labeling is necessary to avoid naming conflicts when multiple local classes are defined within the same scope/method.) The `selfNN`–object represents the currently active `try` block environment together with all of its `recover` handlers in the `recover` handler stack at runtime. On its creation `selfNN` is pushed onto the stack. Therefore, the topmost object of the stack always represents the current active `try` block and/or the current level of `recover` handlers. After having processed the `try` block, `selfNN` is removed from the handler stack by calling its member `remove()`. All of these transformations are illustrated in figure 2.

Beside literally containing the original code of `recover` blocks, the method `raise()` has also another major purpose: to ensure that the *proper* `recover` block gets executed. Therefore a simple check is performed whether the thrown exception is an instance of one of the types that are defined by a `recover` block of this level (`try` block). If so, the respective code (of the `recover` block) is executed, if not, the method `raise()` of the preceding element on the handler stack gets called. This is done by means of the member `caller` which references the preceding element.

To sum things up: At any given time, a call to the method `raise()` of the top element `selfNN` of the handler stack will automatically find and execute the appropriate `recover` block. In order to guarantee that the top element is always accessible, the static member `current` of the class `TryRecoverBlock` always references this top element. Thus, resumption can be achieved by calling — and as the case may be returning from — the method

[1]In order to avoid accidental name clashes with user-defined names, all class, method, and variable names introduced by the precompiler are in reality prefixed with a prefix that is omitted here for the sake of readability.

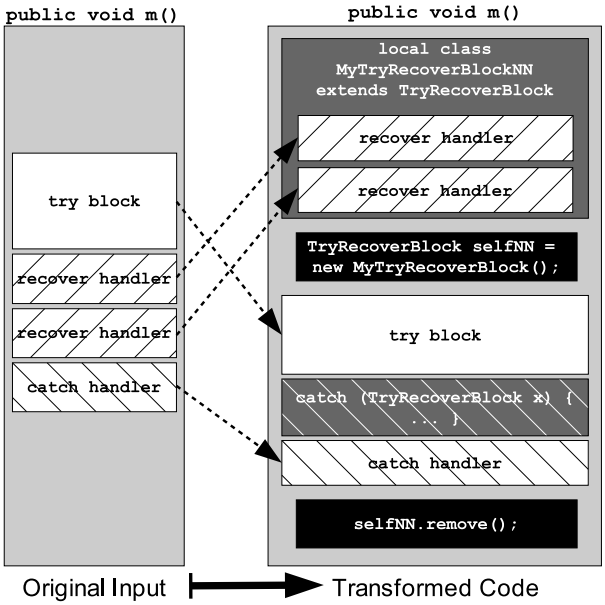

Fig. 2. Sketch of the basic transformation of a `try-recover` construct by the precompiler.

`current.raise()`. In particular this means also that every throw-accept statement can in principle be replaced by `current.raise()`.

But if ordinary termination is wanted, the precompiler has to make sure that execution continues, as with normal Java termination, at the level of the responsible handler. To achieve this, the precompiler adds to every affected `try` block a new `catch` handler for the type `TryRecoverBlock` (which catches also all objects of type `MyTryRecoverBlockNN`). `TryRecoverBlock` itself extends the standard Java class `Error`. Hence, all objects of this class are throwable.
Additionally, the precompiler by default appends a "`throw this;`" statement at the end of every transformed `recover` block's body in the method `raise()`. So, if a `recover` handler does *not* contain or execute a `resume` statement, the corresponding object `selfNN` of the class `MyTryRecoverBlockNN`, which contains the instance–method `raise()` that implements this `recover` block, will automatically be thrown at the end of `raise()`. To ensure that it will be caught and processed by the proper `catch` handler for type `TryRecoverBlock`, a test is made in each of these `catch` handlers, whether the thrown object equals the object `selfNN`, that was created to represent the `try` to which this `catch` handler is attached. If both are equal, the method `raise()`, in which the exception was thrown, implemented a `recover` block of this `try`. In particular, this means that we are at the right `try` block, after which execution has to continue normally again. If the objects are not equal, the method `raise()` does not implement a `recover` block of this `try` block and thus the exception has to be propagated to the preceding `try` block.

3) Transformation of Variables and Real `return` *and/or* `throw` *Statements:* An unacceptable drawback of the use of local classes in Java is that only final variables of the surrounding method can be accessed from within the local class. This would mean for our implementation, that local variables of the enclosing method would not be accessible from within `recover` blocks, as they are from within normal `catch` blocks. In order to make non final variables accessible, the precompiler creates for every non-final variable an *alias variable* which is a final Java array containing a single element, the original variable itself. The precompiler inserts the declaration of these alias variables usually (except for alias variables of parameters) directly after the declaration of the original variable. Once the alias variable has been created, all accesses to the original variable will be replaced by accesses to the respective alias variable — of course, with respect to the scopes of the variables. Alias variables themselves are accessible from within local classes because they are final (arrays). But they still can take different values, because in Java the content of an array can be changed, even if the reference to the array itself is final.

Due to the fact, that `recover` blocks are implemented as methods, real `return` (and `break`), simple `throw` statements (without `accept` blocks) and original `this` references have to be transformed when occurring within a `recover` block, too. For example, if a `return` (`break`) statement would be left literally in a `recover` block, it would end the implementing method `raise()` instead of ending the method which contains the definition of this `recover` block, as the programmer originally intended it. A similar situation arises when simple `throw` statements occur within `recover` blocks. Since the precompiler transforms `recover` blocks into methods, the respective exceptions would be thrown from the wrong level (method) and thus might be handled by incorrect `catch` handlers defined coincidently in a method between `raise()` and the method which defines the intended `catch` handler. Thus, in both cases, the precompiler has to transform the original statements in order to achieve the originally intended effect: By using the original EHM of Java one gets to the proper "level" (of the process stack) first, before being able to execute the *original* `return` or `throw` instruction from there.

V. Conclusion

This paper has described a conceptual model as well as a practical implementation of a resumption mechanism for Java. Even though the termination model seems to be the single best option for a lot of language designers, we do not accept to reject resumption completely, for there are many scenarios where a resumption mechanism provides a superior solution compared to the termination model. In particular, since a resumption mechanism can be implemented and realized rather easily in/for Java, it would be desirable to include it as a regular part of the EHM of a language.

Our approach has — compared to some resumption mechanisms of earlier languages — the advantage, that it allows to pass information from the handling context back to the raising point. This design overcomes some major problems of existing mechansims: Firstly, it allows to determine several ways of how to resume depending on the "solution" that has been chosen by the handler (by specifying different `accept` blocks) and secondly, it provides a way to inform the faulting method about changes in the state of the raising context that have been made by the handler. In that way our mechanism provides a consequent and comprehensible way of resuming execution at the raising point of an exception and hence resumption ceases to be an error–prone form of continuing execution in a possibly inconsistent state of the environment/system. Another very interesting property of the proposed mechanism is that it subsumes the functionality of the original EHM of Java. This means in particular, that the existing `throw` and `catch` constructs could be replaced by the proposed `throw-accept` and `recover-resume` constructs without losing any expressive power of the termination mechanism. In fact, our implementation included the "old" constructs only to stay compatible to existing code. But for future usage, if termination is desired, it can simply be achieved by not including a `resume` statement into a `recover` block, which makes the original constructs/mechanism of Java redundant.

References

[1] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall International, 1997, ISBN 0-13-629155-4.

[2] F. Cristian, "Exception Handling," C.S. and Eng. Dep., University of California, San Diego, Tech. Rep. RJ5724 (57703), 1987.

[3] S. Drew, *et al.*, "Implementing Zero Overhead Exception Handling," Faculty of Information Technology, Queensland University of Technology, Australia, Tech. Rep. 95-12, 1995.

[4] B. Liskov, "A History of CLU," *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 133–147, 1993.

[5] B. Stroustrup, *The C++ Programming Language*, Special ed. Addison-Wesley Publishing Company, 2000, ISBN 0-201-70073-5.

[6] B. Stroustrup and B. Foote, "Why not Resumable Exceptions," Interview Internet, 1996. [Online]. Available: http://cpptips.hyperformix.com/cpptips/term_except

[7] A. Koenig and B. Stroustrup, "Exception Handling for C++ (revised)," in *USENIX C++ Conference Proceedings*. San Francisco, California: The USENIX Association, April 1990, pp. 149–176.

[8] R. Govindarajan, "Exception Handlers in Functional Programming Languages," *Software Engineering*, vol. 19, no. 8, pp. 826–834, 1993.

[9] A. T. Schreiner and B. Kühl, "Exceptions einmal anders [Exceptions the other way round]," *IX*, vol. 11, p. 194, 1999. [Online]. Available: http://www.heise.de/ix/artikel/1999/11/194/

[10] B. Meyer, *Eiffel: The Language*. London,UK: Prentice Hall International, 1992, ISBN 0-13-247925-7.

[11] Paul Graham, *ANSI Common Lisp*, ser. Prentice Hall Series in A.I. Prentice Hall, Inc, 1996, ISBN 0-13-370875-6.

[12] O. Lehrmann Madsen, *et al.*, *Object Oriented Programming in the Beta Programming Language*, ser. ACM Press Books. Addison-Wesley Publishing Company, 1993, ISBN 0-201-62430-3.

[13] A. Gruler, "Flexible Exception Handling in Programming Languages," Master's thesis, University of Ulm, Germany, December 2004. [Online]. Available: http://www.alexander.gruler.org/mastersthesis

[14] S. Viswanadha, *et al.*, "JavaCC: JavaCC Home," Internet, January 2005. [Online]. Available: https://javacc.dev.java.net/