Analysis of the .NET CLR Exception Handling Mechanism

Nicu G. Fruja
Computer Science Department, ETH Zürich
fruja@inf.ethz.ch

Egon Börger
Dipartimento di Informatica, Università di Pisa
boerger@di.unipi.it

ABSTRACT

We provide a complete mathematical model for the exception handling mechanism of the Common Language Runtime (CLR), the virtual machine underlying the interpretation of .NET programs. The goal is to use this rigorous model in the corresponding part of the still-to-be-developed soundness proof for the CLR bytecode verifier.

Keywords
exception handling, .NET CLR, .NET CIL, bytecode

1 INTRODUCTION

This work is part of a larger project [6] which aims at establishing some outstanding properties of C♯ and CLR by mathematical proofs. Examples are the correctness of the bytecode verifier of CLR, the type safety (along the lines of the first author's correctness proof [12] for the definite assignment rules of C♯), the correctness of a general compilation scheme. We try to reuse as much as possible and to extend where necessary similar work which has been done for Java and the Java Virtual Machine (JVM) [15]. As part of this effort, in [8] an abstract interpreter has been developed for C♯, including a thread and memory model [9]; see also [10] for a comparative view of the abstract interpreters for Java and for C♯.

In [7] an abstract model is defined for the CLR virtual machine without the exception handling instructions, but including all the constructs which deal with the interpretation of the procedural, object-oriented and unsafe constructs of .NET compatible languages such as C♯, C++, Visual Basic, VBScript, etc. The reason that we present here a separate model for the exception handling mechanism of CLR is to be found in the numerous non-trivial problems we encountered in an attempt to fill in the missing parts on exception handling in the ECMA standard [1]. Already in JVM the most difficult part for the correctness proof of the bytecode verifier was the one dealing with exception handling (see [15, §16]). This holds in a stronger sense also for CLR. The concrete purposes we are pursuing in this paper are twofold. First we want to define a rigorous ground model for the CLR exception mechanism, to be used as reference model for that part of the still to be developed correctness proof for the bytecode verifier. Secondly we want to clarify the numerous issues concerning exception handling which are left open in the ECMA standard, but which are relevant for a correct understanding of the CLR mechanism. We do not discuss here its design rationale nor any design alternatives.

The ECMA standard for CLR contains only a few yet incomplete paragraphs about the exception handling mechanism. A more detailed description of the mechanism can be found in almost the only document on the CLR exception handling [2]. The CLR mechanism has its origins in the Windows NT Structured Exception Handling (SEH). An interested reader can find all the insights of the SEH in [3]. What we are striving for, the CLR type safety, is proved for a subset of CLR in [4]. However, that approach does not consider the exception handling classified in [4, §4] as a fairly elaborate model that permits a unified view of exceptions in C++, C♯, and other high-level languages. So far, no formal model has been developed for the CLR exception handling. The JVM exception mechanism, which differs a lot from the one of CLR, has been formalized in [16].
We use three different methods to check the faithfulness with respect to CLR of the modeling decisions we had to take where the ECMA standard exhibits deplorable gaps. First of all we made a series of experiments with CLR, some of which are made available in [5] to allow the reader redoing and checking them. We hope that these programs will be of interest to the practitioner and compiler writer, as they show border cases which have to be considered to get a full understanding and definition of exception handling in CLR. Secondly, to provide some authoritative evidence for the correctness of the modeling ideas we were led to by our experiments, over the Fall of 2004 the first author had an electronic discussion with Jonathan Keljo, the CLR Exception System Manager, which essentially confirmed our ideas about the exception mechanism issues left open in the ECMA documents. Last but not least a way is provided to test the internal correctness of the model presented in this paper and its conformance to the experiments with CLR, namely by an executable version of the CLR model, using AsmL. [13]. Upon completion of the AsmL implementation of the entire CLR model the full details will be made available in [14].

Since the focus of this paper is the exception mechanism of CLR, we assume the reader to be knowledgeable about (or at least to have a rough understanding of) CLR. For the sake of precision we refer in this paper without further explanations to the model EXECCLR_N defined in [7], which describes what the machine does upon its "normal" (exception-free) execution. Our model for CLR together with the exception mechanism comes in the form of an Abstract State Machine (ASM) CLR_E.

Since the intuitive understanding of the ASMs machines as pseudo-code over abstract data structures is sufficient for the comprehension of CLR_E, we abstain here from repeating the formal definition of ASMs which can be found in the AsmBook [17]. However, for the readers convenience we summarize here the most important concepts and notations that are used in the ASMs throughout this paper. An abstract state of an ASM is given by a set of dynamic functions. Nullary dynamic functions correspond to ordinary state variables. Formally all functions are total. They may, however, return the special element undefined if they are not defined at an argument. In each step, the machine updates in parallel some of the functions at certain arguments. The updates are programmed using transition rules P, Q with the following meaning:

\[
\begin{align*}
  f(s) &:= t & &\text{update } f \text{ at } s \text{ to } t \\
  \text{if } \varphi \text{ then } P \text{ else } Q & &\text{if } \varphi \text{, then execute } P \text{, else } Q \\
  P &\text{ or } Q & &\text{execute } P \text{ or } Q \text{ in parallel} \\
  \text{let } x = t &\text{ in } P & &\text{assign } t \text{ to } x \text{ and then execute } P \\
  P \text{ seq } Q & &\text{execute } P \text{ and then } Q \\
  P &\text{ seq } Q & &\text{execute } P \text{ or } Q
\end{align*}
\]

Notational conventions In the paper, beside the usual list operations (e.g. push, pop, top, length, \footnote{The "\cdot" denotes the operation append for lists.}) we use a different operation: for a list L, split(L,1) splits off the last element of L. More exactly, split(L,1) is the pair \((L',[x])\) where \(L' : [x] = L\).

The paper is organized as follows. We list in Section 2 a few notations defined in [7] and which are used throughout the rest of the paper. Section 3 gives an overview of the CLR exception handling mechanism. The elements of the formalization are introduced in Section 4. Section 5 defines the so-called StackWalk pass of the exception mechanism. The other two passes, Unwind and Leave are defined in Section 6 and Section 7 respectively. The execution rules of CLR_E are introduced in Section 8. Section 9 concludes.

\section{Preliminaries}

In this section, we summarize briefly the notations introduced in [7] which are relevant for the exception handling mechanism. For detailed description we refer the reader to [7].

A call frame consists of a program counter \(pc : Pc\), local variables addresses \(locAdr : \text{Map}(\text{Local}, \text{Adr})\), arguments addresses \(argAdr : \text{Map}(\text{Arg}, \text{Adr})\), an evaluation stack \(\text{evalStack} : \text{List}(\text{Value})\), and a method reference \(\text{meth} : \text{MRef}\). The frame denotes the currently executed frame. Accordingly, \(pc\) gives the program counter of the current frame, \(locAdr\) the local variables addresses of the current frame, etc.

The stack of call frames is denoted by \(\text{frameStack}\) and is defined as a list of frames. Note that we separate the current frame from the stack of call frames, i.e. \(\text{frame}\) is not contained in \(\text{frameStack}\).

The macros \text{PUSH}FRAME and \text{POP}FRAME are used to push and pop the \text{frame}, respectively.

\begin{align*}
\text{PUSH}FRAME &\equiv \text{push(frameStack,frame)} \\
\text{POP}FRAME &\equiv \text{pop(frameStack)} \\
\text{let} (\text{frameStack}', &\text{=} \text{split(frameStack,1) in} \\
\text{pc} &:= \text{pc}' \\
\text{locAdr} &:= \text{locAdr}' \\
\text{argAdr} &:= \text{argAdr}' \\
\text{evalStack} &:= \text{evalStack}' \\
\text{meth} &:= \text{meth}' \\
\text{frameStack} &:= \text{frameStack}'
\end{align*}

\footnote{In order to simplify the exposition we describe here the \text{evalStack} as a list of values though [7] defines it as a list of pairs from Value \times Type.}
3 THE OVERALL PICTURE

Every time an exception occurs, the control is transferred from “normal” execution (in \texttt{EXECCLR}_E) to a so-called “exception-handling mechanism” which we model as a submachine \texttt{EXECCLR}. To switch from normal execution (read: in mode \texttt{Noswitch}) to this new component, the mode is set to, say, \texttt{switch} := \texttt{ExcMech} which interrupts \texttt{EXECCLR}_E and triggers the execution of \texttt{EXECCLR}. The machine \texttt{EXECCLR}_E is an extension of the exception-handling-free machine \texttt{EXECCLR}_N by a submachine which executes instructions related to exceptions (like \texttt{Throw}, \texttt{Rethrow}, etc.); it will be defined in Fig. 3. Due to the very weak conditions imposed by the ECMA standard on class initialization, the overall structure of \texttt{CLR}_E has to foresee that the initialization of a \texttt{beforefieldinit} class may start at any moment as analyzed in detail in [11]: this explains the definition of \texttt{CLR}_E as a machine which, in the normal execution mode, non-deterministically chooses whether to start a class initialization or to execute the current instruction \texttt{code}(\texttt{pc}) pointed at by the program counter \texttt{pc} (see Fig. 4).

The exception handling mechanism proceeds in two passes. In the first pass, the run-time system runs a “stack walk” searching, in the possibly empty exception handling array associated by \texttt{excHA} : \texttt{Map}([\texttt{MRef}, \texttt{List}(\texttt{Exc})) to the current method, for the first handler that might want to handle the exception:

- a \texttt{catch} handler whose \texttt{type} is a supertype of the \texttt{type} of the exception, or
- a \texttt{filter} handler – to see whether a \texttt{filter} wants to handle an exception, one has first to execute (in the first pass) the code in the filter region: if it returns 1, then it is chosen to handle the exception; if it returns 0, this handler is not good to handle the exception.

Visual Basic and Managed C++ have special \texttt{catch} blocks which can “filter” the exceptions based on the exception type and/or any conditional expression. These are compiled into \texttt{filter} handlers in the

Common Intermediate Language (CIL) bytecode. As we will see, the \texttt{filter} handlers bring a lot of complexity to the exceptions mechanism.

The ECMA standard does not clarify what happens if the execution of the \texttt{filter} or of a method called by it throws an exception. The currently handled exception is known as an \texttt{outer exception} while the newly occurred exception is called an \texttt{inner exception}. As we will see below, the outer exception is not discarded but its context is saved by \texttt{EXECCLR} while the inner exception becomes the outer exception.

If a match is not found in the \texttt{faulting frame}, i.e. the frame where the exception has been raised, the calling method is searched, and so on. This search eventually terminates since the \texttt{excHA} of the \texttt{entrypoint} method has as last entry a so-called \texttt{backstop entry} placed by the operating system. When a match is found, the first pass terminates and in the second pass, called “unwinding of the stack”, \texttt{CLR} walks once more through the stack of call frames to the handler determined in the first pass, but this time executing the \texttt{finally} and \texttt{fault} handlers and popping their frames. It then starts the corresponding exception handler.

The reader might ask why there are two passes, i.e. why the handling mechanism does not proceed in a single pass by executing also the \texttt{finally} and \texttt{fault} handlers. The answer is to be found in the origins of the \texttt{CLR} exception handling mechanism: the two pass model was invented for Windows NT, before the \texttt{CLR} was ever envisioned. There are two advantages of a 2-pass model:

- it allows a \texttt{filter} to update the exception context and to then continue the faulting exception;
- it allows for better debugging, since one can often detect that an exception will go unhandled in the first pass, without any second pass backout disturbing the exception context;

4 THE GLOBAL VIEW OF \texttt{EXECCLR}

In this section, we provide some detail on the elements, functions and predicates needed to turn the overall picture into a rigorous model.

The elements of an exception handling array \texttt{excHA} : \texttt{Map}([\texttt{MRef}, \texttt{List}(\texttt{Exc})) are known as \texttt{handlers} and can be of four kinds. They are elements of a set \texttt{Exc}:

\begin{itemize}
  \item \texttt{catch}
  \item \texttt{filter}
  \item \texttt{try}
  \item \texttt{finally}
\end{itemize}
Any 7-tuple of the above form describes a handler of kind \( clauseKind \) which “protects” the region \( \text{filter} \) that starts at \( \text{tryStart} \) and has the length \( \text{tryLength} \), handles the exception in an area of instructions that starts at \( \text{handlerStart} \) and has the length \( \text{handlerLength} \) – we refer to this area as the \( \text{handler region} \); if the handler is of kind \( \text{catch} \), then the type of exceptions it handles is provided while if the handler is of kind \( \text{filter} \), then the first instruction of the \( \text{filter region} \) is at \( \text{filterStart} \). In case of a \( \text{filter} \) handler, the handler region starting at \( \text{handlerStart} \) immediately follows the \( \text{filter region} \) – consequently we have \( \text{filterStart} < \text{handlerStart} \). We often refer to the sequence of instructions between \( \text{filterStart} \) and \( \text{handlerStart} – 1 \) as the \( \text{filter region} \). We assume that a \( \text{filterStart} \) is defined for a handler if and only if the handler is of kind \( \text{filter} \), otherwise \( \text{filterStart} \) is undefined.

To simplify the further presentation, we define the predicates in Fig. 2 for an instruction located at program counter position \( \text{pos} \in \text{Pc} \) and a handler \( h \in \text{Exc} \). Note that if the predicate \( \text{isInFilter} \) is true, then \( \text{filterStart} \) is defined and therefore \( h \) is of kind \( \text{filter} \). Based on the lexical nesting constraints of protected blocks specified in [1, Partition I,§12.4.2.7], one can prove the following property:

**Disjointness 1** The predicates \( \text{isInTry} \), \( \text{isInHandler} \) and \( \text{isInFilter} \) are pairwise disjoint.

We assume all the constraints concerning the lexical nesting of handlers specified in the standard [1, Partition I,§12.4.2.7]. The ECMA standard [1, Partition I,§12.4.2.5] ordering assumption on handlers is:

**Ordering assumption** If handlers are nested, the most deeply nested try blocks shall come in the exception handling array before the try blocks that enclose them.

**Only one handler region per try block?** The ECMA standard specifies in [1, Partition I,§12.4.2] that a single \( \text{try} \) block shall have exactly one handler region associated with it. But the IL assembler \( \text{ilasm} \) does accept also \( \text{try} \) blocks with more than one \( \text{catch} \) handler block. This discrepancy is solved if we assume that every \( \text{try} \) block has more than one \( \text{catch} \) block which is accepted by the \( \text{ilasm} \) is translated in a semantics preserving way as follows:

To handle an exception, the \( \text{EXCCLR} \) needs to record:

- the exception reference \( \text{exc} \),
- the handling \( \text{pass} \),
- a \( \text{stackCursor} \) – i.e. the position currently reached in the stack of call frames (a frame) and in the exception handling array (an index in \( \text{excHA} \)),
- the suitable \( \text{handler} \) determined at the end of the \( \text{StackWalk} \) pass (if any) is the handler that is going to handle the exception in the pass \( \text{Unwind} \) – until the end of the \( \text{StackWalk} \) pass, \( \text{handler} \) is undefined.

According to the ECMA standard, every normal execution of a \( \text{try} \) block or a \( \text{catch/filer} \) handler region must end with a \( \text{Leave(pos)} \) instruction. When doing this, \( \text{EXCCLR} \) has to record the current \( \text{pass} \) and \( \text{stackCursor} \) together with the \( \text{target} \) up to which every included \( \text{finally} \) code has to be executed.

We list some constraints which will be needed below to understand the treatment of these \( \text{Leave} \) instructions.
It is not legal to exit with a try. It is legal to exit with a block, so long as a catch block protects the try block protects the a filter region con-

F or every type t, the predicates

StackWalk 5 THE pass at the Leave continue its work in the invoker frame or end its

Leaving) the corresponding handler; otherwise it will Leave the search position

rent exception in this frame. Such a handler exists if

Unwind or to execute (when reached the last element of the handlers array

excHA of the corresponding method

The nesting of passes determines EXCCLR to maintain an initially empty stack of exception or leave records for the passes that are still to be performed.

passRecStack : List(ExcRec ∪ LeaveRec)
passRecStack = []

In the initial state of EXCCLR, there is no pass to be executed, i.e. pass = undef.

We can now summarize the overall behavior of EXCCLR, which is defined in Fig. 3 and analyzed in detail in the following sections, by saying that if there is a handler in the frame defined by stackCursor, then EXCCLR will try to find (when StackWalking) or to execute (when Unwinding) or to leave (when Leaving) the corresponding handler; otherwise it will continue its work in the invoker frame or end its Leave pass at the target.

5 THE StackWalk PASS

During a StackWalk pass, EXCCLR starts in the current frame to search for a suitable handler of the current exception in this frame. Such a handler exists if the search position \( n \) in the current frame has not yet reached the last element of the handlers array excHA of the corresponding method \( m \).

existsHanWithinFrame((..., ..., ..., n, m), n) ⇐ n < length(excHA(m))

If there are no (more) handlers in the frame pointed to by stackCursor, then the search has to be continu-

ued at the invoker frame. This means to reset the stackCursor to point to the invoker frame.

SEARCHINVFRAME(f) ≡

let f’ = frameStack · [f] in
RESET(stackCursor, f’)

There are three groups of possible handlers \( h \) EXCCLR is looking for in a given frame during its StackWalk:

• a catch handler whose try block protects the program counter \( pc \) of the frame pointed at by stackCursor and whose type is a supertype of the exception type:

matchCatch(pos, t, h) ⇐ isInTry(pos, h) ∧ clauseKind(h) = catch ∧ t ⪯ type(h)

• a filter handler whose try block protects the pc of the frame pointed at by stackCursor;

matchFilter(pos, h) ⇐ isInTry(pos, h) ∧ clauseKind(h) = filter

• a filter handler whose filter region contains \( pc \) of the frame pointed at by stackCursor.

This corresponds to an outer exception and will be described in more detail below.

The order of the if clauses in the let statement from the rule StackWalk is not important. This is justified by the following property:

Disjointness 2 For every type \( t \), the predicates matchCatch \( t \), matchFilter and isInFilter are pairwise disjoint\footnote{By matchCatch \( t \) we understand the predicate defined by the set \{ (pos, h) | matchCatch(pos, t, h) \}.}

The above property can be easily proved using the definitions of the three predicates and the property Disjointness\footnote{By matchCatch \( t \) we understand the predicate defined by the set \{ (pos, h) | matchCatch(pos, t, h) \}.}

If the handler pointed to by the stackCursor, namely hanWithinFrame((..., ..., ..., n, m), n) = excHA(m)(n), is not of any of the above types, the stackCursor is incremented to point to the next handler in the excHA:
The Ordering assumption stated in Section 4 and the lexical nesting constraints stated in [1] Part 1.12.4.2.7] ensure that, if the stackCursor points to a handler of one of the above types, then this handler is the first handler in the exception handling array (starting at the position indicated in the stackCursor) of any of the above types.

If the handler pointed to by the stackCursor is a matching 

\[catch, then this handler becomes the \text{handler} to handle the exception in the pass Unwind.\]

The stackCursor is reset to be reused for the Unwind pass: it shall point to the faulting frame, i.e. the current frame. Note that during \text{StackWalk}, frame always points to the faulting frame except in case a filter region is executed. However, the frame built to execute a filter is never searched for a handler corresponding to the current exception.

If the handler is a filter, then by means of \text{EXECFilter} its filter region is executed. The execution is performed in a separate frame constructed especially for this purpose. However this important detail is omitted by the ECMA standard [1]. The current frame becomes the frame for executing the filter region. The faulting exception frame is pushed on the frameStack. The current frame points now to the method, local variables and arguments of the frame in which stackCursor is, it has the exception reference on the evaluation stack evalStack and the program counter pc set to the beginning filterStart of the filter region. The switch is set to Noswitch in order to pass the control to the normal machine EXECLR_E.

---

### FORMULA

\[\text{FoundHandler} \equiv\]

\[
\text{pass} := \text{Unwind} \\
\text{handler} := \text{stackCursor}
\]

\[\text{RESET}(s,f) \equiv s := (f,0)\]
following cases are to be considered:

- if the exception is taken care of in the filter region, i.e. it is successfully handled by a catch/filter handler or it is aborted because it occurred in yet another filter region of a nested handler (see the isInFilter clause), then the given filter region continues executing normally (after the exception has been taken care of);

- if the exception is not taken care of in the filter region, then the exception is not propagated further, but its StackWalk is exited (see Fig. 3). The exception will be discarded but only after the EXCCLR runs its Unwind pass to execute all the finally and fault handlers (see Tests 6, 8 and 9 in [5]).

6 THE Unwind PASS

As soon as the pass StackWalk terminates, the EXCCLR starts the Unwind pass with the stackCursor pointing to the faulting exception frame. Starting there one has to walk down to the handler determined in the StackWalk, executing on the way every finally/fault handler region. This happens also in case handler is undef. When Unwinding, the EXCCLR searches for:

- the matching target handler, i.e. the handler determined at the end of the StackWalk pass (if any) – handler can be undef if the search in the StackWalk has been exited because the exception was thrown in a filter region. Also the two handler and stackCursor frames in question have to coincide. We say that two frames are the same if the address arrays of their local variables and arguments as well their method names coincide.

\[
\text{EXITINNEREXC} \equiv \\
\text{pass} := \text{Unwind} \\
\text{RESET} (\text{StackCursor.frame})
\]

- a matching finally/fault handler whose associated try block protects the pc;

- a handler whose handler region contains pc;

- a filter handler whose filter region contains pc;

The order of the last three if clauses in the let statement from the rule Unwind is not important. It matters only that the first clause is guarded by matchTargetHan.

Disjointness 3 The following predicates are pairwise disjoint: matchFinFault, isInHandler and isInFilter.

The property can be proved using the definitions of the predicates and the property Disjointness 1.

The Ordering assumption in Section 4 and the lexical nesting constraints given in Partition 1,[12.4.2.7] ensure that, if the stackCursor points to a handler of one of the above types, then this handler is the first handler in the exception handling array (starting at the position indicated in the stackCursor) of any of the above types.

If the handler pointed to by the stackCursor is the handler found in the StackWalk, its handler region is executed through EXECHan: the pc is set to the beginning of the handler region, the exception reference is loaded on the evaluation stack (when EXECHan is applied for executing finally/fault handler regions the current exception is not pushed on evalStack) and the control switches to EXECCLR.

\[
\text{EXECHan}(h) \equiv \\
\text{pc} := \text{handlerStart}(h) \\
\text{evalStack} := \\
\text{if clauseKind}(h) \in \{\text{catch, filter}\} \text{ then} \\
\text{[exc]} \\
\text{else} \\
\text{[]} \\
\text{switch} := \text{Noswitch}
\]

If the handler pointed to by the stackCursor is a matching finally/fault handler, its handler region is executed with initially empty evaluation stack. At the same time, the stackCursor is incremented through GoToNXTHan.

Let us assume that the handler pointed to by stackCursor is an arbitrary handler whose handler region contains pc.

Exceptions in handler region? The ECMA standard does not specify what should happen if an exception is raised in a handler region. The experimentation in [5] can be resumed by the following rules of thumb for exceptions thrown in a handler region similarly to the case of nested exceptions in filter code:

- if the exception is taken care of in the handler region, i.e. it is successfully handled by a
catch/filter handler or it is discarded (because it occurred in a filter region of a nested handler), then the handler region continues executing normally (after the exception is taken care of);

- if the exception is not taken care of in the handler region, i.e., escapes the handler region, then
  - the previous pass of exccLR is aborted through \texttt{AbortPrevPassRec};

\[
\text{AbortPrevPassRec} \equiv \text{pop(passRecStack)}
\]

- the exception is propagated further, i.e. the Unwind pass continues via \texttt{goto\_nxt\_han} (see Fig. 3) which sets the stackCursor to the next handler in excHA.

The execution of a handler region can occur only when excCLR runs in the Unwind and Leave passes: in Unwind handler regions of any kind are executed while in Leave only \texttt{finally} handler regions are executed. If the raised exception occurred while excCLR runs an Unwind pass for handling an outer exception, the Unwind pass of the outer exception is stopped and the corresponding pass record is popped from passRecStack (see Tests 1, 3 and 4 in [5]). If the exception has been thrown while excCLR runs a Leave pass for executing finally handlers on the way from a Leave instruction to its target, then this pass is stopped and its associated pass record is popped off passRecStack (see Test 2 in [5]).

In this way an exception can go “unhandled” without taking down the process, namely if an outer exception goes unhandled, but an inner exception is successfully handled (see the second case of the preceding case distinction).

If the handler pointed to by the stackCursor is a filter handler whose filter region contains \texttt{pc}, then the current (inner) exception is aborted and the filter considered as not providing a handler for the outer exception. So there is no way to exit a filter region with an exception. This ensures that the frame built by EXEC\texttt{FILTER} for executing a filter region is used only for this purpose. The handling of the outer exception is continued through \texttt{continue\_outer\_exc} (see Fig. 3) which pops the frame built for executing the filter region, pops from the passRecStack the pass record corresponding to the inner exception and reestablishes the pass context of the outer exception, but with the stackCursor pointing to the handler following the just inspected filter handler. The updates of the stackCursor in POP\texttt{REC} and GOTO\texttt{NXT\_HAN} are done sequentially such that the update in GOTO\texttt{NXT\_HAN} overwrites the update in POP\texttt{REC}.

\[
\text{ContinueOuterExc} \equiv \begin{array}{l}
\text{PopFrame} \\
\text{PopRec seq GotoNxtHan}
\end{array}
\]

\[
\text{PopRec} \equiv \begin{array}{l}
\text{if passRecStack} = [] \text{ then} \\
\text{SetRecUnDef} \\
\text{switch} := \text{Noswitch} \\
\text{else let (passRecStack', [r]) = split(passRecStack, 1) in} \\
\text{if} \; r \in \text{ExcRec} \text{ then} \\
\text{let (exc', pass', stackCursor', handler') = r in} \\
\text{exc} := \text{exc}' \\
\text{pass} := \text{pass}' \\
\text{stackCursor} := \text{stackCursor}' \\
\text{handler} := \text{handler}' \\
\text{if} \; r \in \text{LeaveRec} \text{ then} \\
\text{let (pass', stackCursor', target') = r in} \\
\text{pass} := \text{pass}' \\
\text{stackCursor} := \text{stackCursor}' \\
\text{target} := \text{target}' \\
\text{passRecStack} := \text{passRecStack'} \\
\text{SetRecUnDef} \equiv \\
\text{exc} := \text{undef} \\
\text{pass} := \text{undef} \\
\text{stackCursor} := \text{undef} \\
\text{target} := \text{undef} \\
\text{handler} := \text{undef}
\end{array}
\]

If the handler pointed to by the stackCursor is not of any of the above types, the stackCursor is incremented to point to the next handler in the excHA.

If the Unwind pass exhausted all the handlers in the frame indicated in stackCursor, then the current frame is popped from frameStack and the Unwind pass continues in the invoker frame of the current frame.

Exceptions in class initializers? If an exception occurs in a class initializer, \texttt{cctor}, then the class shall be marked as being in a specific erroneous state and a TypeInitializationException is thrown. This means that an exception can and will escape the body of an initializer only by the specific exception TypeInitializationException. Any further attempt to access the corresponding class in the current application domain will throw the same TypeInitializationException object. Unfortunately, these details are not specified by the ECMA standard but it seems to correspond to the actual CLR implementation and it complies with the related specification for C# in the ECMA standard (see Test 7 in [5]). Therefore we assume that the code sequence of every \texttt{cctor} is embedded into
a catch handler. This catch handler catches exceptions of type Object, i.e., any exception, occurring in .ctor, discards it, creates an object of type TypeInitializationException and throws the new exception.

7 THE Leave PASS

The EXCCLR machine gets into the Leave pass when EXCCLR_E executes a Leave instruction upon the normal termination of a try block or of a catch/filter handler region. One has to execute the handler regions of all finally handlers on the way from the Leave instruction to the instruction whose program counter is given by the Leave target parameter. The stackCursor used in the Leave pass is initialized by the Leave instruction. In the Leave pass, the EXCCLR machine searches for

- finally handlers that are “on the way” from the pc to the target,
- real handlers, i.e., catch/filter handlers that are “on the way” from the pc to the target – more details are given below.

If the handler pointed to by stackCursor is a finally handler on the way from pc to target position of the current Leave pass record, then the handler region of this handler is executed (see Fig. 3). If the stackCursor points to a catch/filter handler on the way from pc to target, then the previous pass record on passRecStack is discarded (see Fig. 3). The discarded record can only be referring to an unwind pass for handling an exception. By discarding this record, the mechanism terminates the handling of the corresponding exception.

For each handler EXCCLR inspects also the next handler in excHA. When the handlers in the current method are exhausted, pc is set to target, the context of the previous pass record on passRecStack is reestablished and the control is passed to normal EXCCLR_E execution (see Fig. 3).

8 THE RULES OF EXCCLR_E

The rules of EXCCLR_E in Fig. 4 specify the effect of the CIL instructions related to exceptions. Each of these rules transfers the control to EXCCLR. Throw pops the topmost evaluation stack element (see Remark below), which is supposed to be an exception reference. It loads on EXCCLR the pass record associated to the given exception: the stackCursor is initialized by the current frame and 0. If the exception mechanism is already working in a pass, i.e., pass ≠ undef, then the current pass record is pushed on passRecStack.

If the exception reference popped from the evalStack by the Throw instruction is null, a NullReferenceException is thrown. For a given class c, the macro RAISE(c) is defined by the following code template.

This macro can be viewed as a static method defined in class Object. Calling the macro is then like invoking the corresponding method.

The ECMA standard states in [1, Partition III, §4.23] that the Rethrow instruction is only permitted within the body of a catch handler. However, the same instruction is allowed also within a handler region of a filter (see Test 5 in [5]) even if this does not

---

In the real CLR implementation, the exception thrown in .ctor is embedded as an inner exception in the TypeInitializationException. We do not model this aspect here.

---

The NewObj instruction called with an instance constructor c::ctor creates a new object of class c and then calls the constructor .ctor.
match the previous statement. It throws the same exception reference that was caught by this handler, i.e. the current exception exc of EXCCLR. Formally, this means that the pass record associated to exc is loaded on EXCCLR.

In a filter region, there should be exactly one EndFilter instruction. This has to be the last instruction in the filter region. EndFilter takes an integer val from the stack that is supposed to be either 0 or 1. In the ECMA standard, 0 and 1 are assimilated with “continue search” and “execute handler”, respectively. There is a discrepancy between [1] Partition I,[2],[3,4] which states Execution cannot be resumed at the location of the exception, except with a user-filtered handler and [1] Partition III,[3,4] which states that the only possible return values from the filter are “exception_continue_search”(0) and “exception_execute_handler”(1). In other words, resumable exceptions are not (yet) supported contradicting Partition I.

If val is 1, then the filter handler to which EndFilter corresponds becomes the handler to handle the current exception in the pass Unwind. Remember that the filter handler is the handler pointed to by the stackCursor. The stackCursor is reset to be used for the pass Unwind: it will point into the topmost frame on frameStack which is actually the faulting frame. If val is 0, the stackCursor is incremented to point to the handler following our filter handler. Independent of val, the current frame is discarded to reestablish the context of the faulting frame. Note that we do not explicitly pop the val from the evalStack since anyway the global dynamic function evalStack is updated anyway in the next step through PopFrame to the evalStack’ of the faulting frame.

The EndFinally instruction terminates the execution of the handler region of a finally/fault handler. It empties the evalStack and transfers the control to EXCCLR. A Leave instruction empties the evalStack and loads on EXCCLR a pass record corresponding to a Leave pass.

Remark The reader might ask why the instructions Throw, Rethrow and EndFilter do not set the evalStack. The reason is that this set up, i.e. the emptying of the evalStack, is supposed to be either a side-effect (the case of the Throw and Rethrow instructions) or ensured for a correct CIL (the case of the EndFilter instruction). Thus, the Throw and Rethrow instructions pass the control to EXCCLR which, in a next step, will execute a catch/finally/fault handler region or a filter code or propagates the exception in another frame. All these “events” will “clear” the evalStack. In case of EndFilter, the evalStack must contain exactly one item (an int32 which is popped off by EndFilter). Note that this has to be checked by the bytecode verifier and not ensured by the exception handling mechanism.

9 CONCLUSION

We have defined an abstract model for the CLR exception handling mechanism. On one hand, this paper has laid the ground to mathematically prove the correctness of the CLR bytecode verifier. On the other hand, through the analysis of the mechanism, we discovered a few gaps in the ECMA standard for CLR. Our model fills in these gaps and precisely specifies the behavior of the mechanism in all the subtle but critical cases.

10 ACKNOWLEDGMENT

We are thankful to Jonathan Keljo for the useful discussion.

References

2003.

One can formally prove that there is such a “step” in the further run of the EXCCLR.


