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 why 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 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 one of very few existing documents 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, 15].

.NET Technologies'2005 *Conference proceedings,*
ISBN 80*-75832,/0,0*

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 to redo and check 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 [18]. 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 reader's 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 *undef* 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:

$f(s) := t$	update f at s to t
if φ **then** P **else** Q	if φ, then execute P, else Q
$P\ Q$	execute P and Q in parallel
let $x = t$ **in** P	assign t to x and then execute P
P **seq** Q	execute P and then Q
P **or** Q	execute P or Q

Notational conventions In the paper, beside the usual list operations (e.g. *push*, *pop*, *top*, *length*, $\cdot$)[1], 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 $L' \cdot [x] = L$.

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.

2 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* : $Map(Local, Adr)$, arguments addresses *argAdr* : $Map(Arg, Adr)$, an evaluation stack[2] *evalStack* : $List(Value)$, and a method reference *meth* : *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 *frameStack* and is defined as a list of frames. Note that we separate the current frame from the stack of call frames, i.e. *frame* is not contained in *frameStack*.

The macros PUSHFRAME and POPFRAME are used to push and pop the *frame*, respectively.

```
PUSHFRAME ≡ push(frameStack, frame)

POPFRAME ≡
  let (frameStack',
         [(pc', locAdr', argAdr', evalStack', meth')])
       = split(frameStack, 1) in
     pc         := pc'
     locAdr     := locAdr'
     argAdr     := argAdr'
     evalStack  := evalStack'
     meth       := meth'
     frameStack := frameStack'
```

[1] The "$\cdot$" denotes the operation *append* for lists.

[2] In order to simplify the exposition we describe here the *evalStack* as a list of values though [7] defines it as a list of pairs from $Value \times Type$.

Fig. 1 The CLR_E machine

$\mathrm{CLR}_E \equiv$
  if *switch* $=$ *ExcMech* **then**
    EXCCLR
  elseif *switch* $=$ *Noswitch* **then**
    INITIALIZECLASS **or** $\mathrm{EXECCLR}_E(code(pc))$

3 THE OVERALL PICTURE

Every time an exception occurs, the control is transferred from "normal" execution (in $\mathrm{EXECCLR}_E$) to a so-called "exception handling mechanism" which we model as a submachine EXCCLR. To switch from normal execution (read: in mode *Noswitch*) to this new component, the mode is set to, say, *switch* := *ExcMech* which interrupts $\mathrm{EXECCLR}_E$ and triggers the execution of EXCCLR. The machine $\mathrm{EXECCLR}_E$ is an extension of the exception-handling-free machine $\mathrm{EXECCLR}_N$ by a submachine which executes instructions related to exceptions (like *Throw*, *Rethrow*, etc.); it will be defined in Fig. 4. Due to the very weak conditions imposed by the ECMA standard on class initialization, the overall structure of CLR_E has to foresee that the initialization of a `beforefieldinit`[3] class may start at any moment as analyzed in detail in [11]; this explains the definition of 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 *code*(*pc*) pointed at by the program counter *pc* (see Fig. 1).

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 *excHA* : $Map(MRef, List(Exc))$ to the current method, for the first handler that might want to handle the exception:

- a `catch` handler whose *type* is a supertype of the type of the exception, or
- a `filter` handler – to see whether a `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 `catch` blocks which can "filter" the exceptions based on the exception type and / or any conditional expression. These are compiled into `filter` handlers in the Common Intermediate Language (CIL) bytecode. As we will see, the `filter` handlers bring a lot of complexity to the exceptions mechanism.

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

If a match is not found in the *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 *excHA* of the `entrypoint` method has as last entry a so-called *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", CLR walks once more through the stack of call frames to the handler determined in the first pass, but this time executing the `finally` and `fault`[4] 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 `finally` and `fault` handlers. The answer is to be found in the origins of the CLR exception handling mechanism: the two pass model was invented for Windows NT, before the CLR was ever envisioned. There are two advantages of a 2-pass model:

- it allows a `filter` to update the exception context and 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 EXCCLR

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 *excHA* : $Map(MRef, List(Exc))$ are known as *handlers* and can be of four kinds. They are elements of a set *Exc*:

[3] The ECMA standard states in [1, Partition I, §8.9.5] that, if a class is marked `beforefieldinit`, then the class initializer method is executed *at any time before* the first access to any static field defined for that class.

[4] Currently, no language (other than CIL) exposes `fault` handlers directly. A `fault` handler is simply a `finally` handler that only executes in the exceptional case.

$$
\begin{array}{lll}
\textit{ClauseKind} = \texttt{catch} \mid \texttt{filter} \\
\qquad\qquad \mid \texttt{finally} \mid \texttt{fault} \\
\textit{Exc} = \textit{Exc}\,(& \textit{clauseKind} & : \textit{ClauseKind} \\
& \textit{tryStart} & : \textit{Pc} \\
& \textit{tryLength} & : \mathbb{N} \\
& \textit{handlerStart} & : \textit{Pc} \\
& \textit{handlerLength} & : \mathbb{N} \\
& \textit{type} & : \textit{ObjClass} \\
& \textit{filterStart} & : \textit{Pc}\ )
\end{array}
$$

Any 7-tuple of the above form describes a handler of kind *clauseKind* which "protects" the region[5] that starts at *tryStart* and has the length *tryLength*, handles the exception in an area of instructions that starts at *handlerStart* and has the length *handlerLength* – we refer to this area as the *handler region*; if the handler is of kind `catch`, then the *type* of exceptions it handles is provided, whereas if the handler is of kind `filter` then the first instruction of the `filter` *region* is at *filterStart*. In case of a `filter` handler, the handler region starting at *handlerStart* immediately follows the `filter` region – consequently we have $\textit{filterStart} < \textit{handlerStart}$. We often refer to the sequence of instructions between *filterStart* and $\textit{handlerStart} - 1$ as the `filter` *region*. We assume that a *filterStart* is defined for a handler if and only if the handler is of kind `filter`, otherwise *filterStart* is undefined.
To simplify the further presentation, we define the predicates in Fig. 2 for an instruction located at program counter position $\textit{pos} \in \textit{Pc}$ and a handler $h \in \textit{Exc}$. Note that if the predicate *isInFilter* is true, then *filterStart* is defined and therefore h is of kind `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 isInTry, isInHandler and 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 `try` block shall have exactly one handler region associated with it. But the IL assembler `ilasm` does accept also `try` blocks with more than one `catch` handler block. This discrepancy is solved if we assume that every `try` block with more than one `catch` block which is accepted by the `ilasm` is translated in a semantics-preserving way as follows:

$$
\begin{array}{l}
\texttt{.try}\ \{ \\
\quad \textit{block} \\
\}\ \texttt{catch}\ \textit{block}_1 \\
\quad \texttt{catch}\ \textit{block}_2
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
\texttt{.try}\ \{ \\
\quad \texttt{.try}\ \{ \\
\qquad \textit{block} \\
\quad \}\ \texttt{catch}\ \textit{block}_1 \\
\}\ \texttt{catch}\ \textit{block}_2
\end{array}
$$

To handle an exception, the EXCCLR needs to record:

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

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

$$
\begin{array}{lll}
\textit{ExcRec} = \\
\textit{ExcRec}\,(& \textit{exc} & : \textit{ObjRef} \\
& \textit{pass} & : \{\textit{StackWalk}, \textit{Unwind}\} \\
& \textit{stackCursor} & : \textit{Frame} \times \mathbb{N} \\
& \textit{handler} & : \textit{Frame} \times \mathbb{N}\ ) \\
\textit{LeaveRec} = \\
\textit{LeaveRec}\,(& \textit{pass} & : \{\textit{Leave}\} \\
& \textit{stackCursor} & : \textit{Frame} \times \mathbb{N} \\
& \textit{target} & : \textit{Pc}\ )
\end{array}
$$

We list some constraints which will be needed below to understand the treatment of these *Leave* instructions.

[5] We will refer to this region as *protected region* or `try` block.

Fig. 2 The predicates *isInTry*, *isInHandler* and *isInFilter*

$$
\begin{array}{lcl}
\mathit{isInTry}(\mathit{pos}, h) & \Leftrightarrow & \mathit{tryStart}(h) \leq \mathit{pos} < \mathit{tryStart}(h) + \mathit{tryLength}(h) \\
\mathit{isInHandler}(\mathit{pos}, h) & \Leftrightarrow & \mathit{handlerStart}(h) \leq \mathit{pos} < \mathit{handlerStart}(h) + \mathit{handlerLength}(h) \\
\mathit{isInFilter}(\mathit{pos}, h) & \Leftrightarrow & \mathit{filterStart}(h) \leq \mathit{pos} < \mathit{handlerStart}(h)
\end{array}
$$

Syntactic constraints:

1. It is not legal to exit with a *Leave* instruction a `filter` region, a `finally`/`fault` handler region.
2. It is not legal to branch with a *Leave* instruction into a handler region from outside the region.
3. It is legal to exit with a *Leave* a `catch` handler region and branch to any instruction within the associated `try` block, so long as that branch target is not protected by yet another `try` block.

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.

$$
\begin{array}{l}
\mathit{passRecStack} : \mathit{List}(\mathit{ExcRec} \cup \mathit{LeaveRec}) \\
\mathit{passRecStack} = [\,]
\end{array}
$$

In the initial state of EXCCLR, there is no pass to be executed, i.e. $\mathit{pass} = \mathit{undef}$.

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 *StackWalk*ing) or to execute (when *Unwind*ing) or to leave (when *Leave*ing) 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.

$$
\begin{array}{l}
\mathit{existsHanWithinFrame}((\_, \_, \_, \_, \_, m), n) \Leftrightarrow \\
\quad n < \mathit{length}(\mathit{excHA}(m))
\end{array}
$$

If there are no (more) handlers in the frame pointed to by *stackCursor*, then the search has to be continued at the invoker frame. This means to reset the *stackCursor* to point to the invoker frame.

$$
\begin{array}{l}
\textsc{SearchInvFrame}(f) \equiv \\
\quad \textbf{let } \_ \cdot [f', f] \cdot \_ = \mathit{frameStack} \cdot [\mathit{frame}] \textbf{ in} \\
\quad\quad \textsc{Reset}(\mathit{stackCursor}, f')
\end{array}
$$

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;

$$
\begin{array}{l}
\mathit{matchCatch}(\mathit{pos}, t, h) \Leftrightarrow \\
\quad \mathit{isInTry}(\mathit{pos}, h) \wedge \mathit{clauseKind}(h) = \texttt{catch} \wedge \\
\quad t \preceq \mathit{type}(h)
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{matchFilter}(\mathit{pos}, h) \Leftrightarrow \\
\quad \mathit{isInTry}(\mathit{pos}, h) \wedge \mathit{clauseKind}(h) = \texttt{filter}
\end{array}
$$

- 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 $\mathit{matchCatch}^t$, matchFilter and isInFilter are pairwise disjoint*[6].

The above property can be easily proved using the definitions of the three predicates and the property *Disjointness* 1.

If the handler pointed to by the *stackCursor*, namely $\mathit{hanWithinFrame}((\_, \_, \_, \_, \_, m), n) = \mathit{excHA}(m)(n)$, is not of any of the above types, the *stackCursor* is incremented to point to the next handler in the *excHA*:

[6] By $\mathit{matchCatch}^t$ we understand the predicate defined by the set $\{(\mathit{pos}, h) \mid \mathit{matchCatch(pos,t,h)}\}$.

GoToNxtHan ≡ *stackCursor* := $(f, n+1)$
where *stackCursor* = (f, n)

The *Ordering assumption* stated in Section 4 and the lexical nesting constraints stated in [1, Partition I,§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[7] `catch` then this handler becomes the *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 *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.

FoundHandler ≡
  pass := *Unwind*
  handler := *stackCursor*

Reset(s, f) ≡ $s := (f, 0)$

If the handler is a `filter` then by means of 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 currently-to-be-executed 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 ExecCLR$_E$.

[7] We use the *actualTypeOf* function defined in [7] to determine the run-time type of the exception.

Fig. 3 The exception handling machine ExcCLR

```
ExcCLR ≡
  match pass
    StackWalk →
      if existsHanWithinFrame(stackCursor) then
        let h = hanWithinFrame(stackCursor) in
          if matchCatch(pos, actualTypeOf(exc), h) then
            FoundHandler
            Reset(stackCursor, frame)
          elseif matchFilter(pos, h) then ExecFilter(h)
          elseif isInFilter(pos, h) then ExitInnerExc
          else GoToNxtHan
      else SearchInvFrame(f)
      where stackCursor = (f, _) and f = (pos, _, _, _, _)

    Unwind →
      if existsHanWithinFrame(stackCursor) then
        let h = hanWithinFrame(stackCursor) in
          if matchTargetHan(handler, stackCursor) then
            ExecHan(h)
          elseif matchFinFault(pc, h) then
            ExecHan(h)
            GoToNxtHan
          elseif isInHandler(pc, h) then
            AbortPrevPassRec
            GoToNxtHan
          elseif isInFilter(pc, h) then
            ContinueOuterExc
          else GoToNxtHan
      else
        PopFrame
        SearchInvFrame(frame)

    Leave →
      if existsHanWithinFrame(stackCursor) then
        let h = hanWithinFrame(stackCursor) in
          if isFinFromTo(h, pc, target) then
            ExecHan(h)
          if isRealHanFromTo(h, pc, target) then
            AbortPrevPassRec
          GoToNxtHan
      else
        pc := target
        PopRec
        switch := Noswitch
```

ExecFilter(h) ≡
  pc := *filterStart*(h)
  evalStack := [*exc*]
  locAdr := *locAdr*′
  argAdr := *argAdr*′
  meth := *meth*′
  PushFrame
  switch := *Noswitch*
  where *stackCursor* =
    $((\_, locAdr', argAdr', \_, meth'), \_)$

Exceptions in `filter` region? It is not documented in the ECMA standard what happens if an (inner) exception is thrown while executing the `filter` region during the *StackWalk* pass of an outer exception. The

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 occured 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]).

EXITINNEREXC ≡
  pass := *Unwind*
  RESET(*stackCursor*, *frame*)

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 *Unwind*ing, 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 as their method names coincide.

$$matchTargetHan((f1,n1),(f2,n2)) \Leftrightarrow sameFrame(f1,f2) \wedge n1 = n2$$

$$sameFrame(f1,f2) \Leftrightarrow pr_i(f1) = pr_i(f2), \forall i \in \{2,3,5\}$$

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

$$matchFinFault(pos,h) \Leftrightarrow isInTry(pos,h) \wedge clauseKind(h) \in \{\texttt{finally}, \texttt{fault}\}$$

- 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 only matters 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 [1, Partition I,§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_E.

EXECHAN(*h*) ≡
  pc := *handlerStart*(*h*)
  evalStack :=
    if *clauseKind*(*h*) ∈ {`catch`, `filter`} **then**
      [*exc*]
    else
      [ ]
  switch := *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 occured 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 ABORTPREVPASSREC;

  ABORTPREVPASSREC ≡ *pop*(*passRecStack*)

  – the exception is propagated further, i.e. the *Unwind* pass continues via GOTONXTHAN (see Fig. 3) which sets the *stackCursor* to the next handler in *excHA*.

The execution of a handler region can only occur when EXCCLR runs in the *Unwind* and *Leave* passes: in *Unwind* handler regions of any kind are executed whereas in *Leave* only `finally` handler regions are executed. If the raised exception occured 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 *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 EXECFILTER for executing a `filter` region is used only for this purpose. The handling of the outer exception is continued through CONTINUEOUTEREXC (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 POPREC and GOTONXTHAN are done **seq**uentially such that the update in GOTONXTHAN overwrites the update in POPREC.

CONTINUEOUTEREXC ≡
  POPFRAME
  POPREC **seq** GOTONXTHAN

POPREC ≡
  if *passRecStack* = [ ] **then**
    SETRECUNDEF
    switch := *Noswitch*
  else let (*passRecStack*′, [*r*]) =
        split(*passRecStack*, 1) **in**
    if *r* ∈ *ExcRec* **then**
      let (*exc*′, *pass*′, *stackCursor*′, *handler*′) = *r* **in**
        exc := *exc*′
        pass := *pass*′
        stackCursor := *stackCursor*′
        handler := *handler*′
    if *r* ∈ *LeaveRec* **then**
      let (*pass*′, *stackCursor*′, *target*′) = *r* **in**
        pass := *pass*′
        stackCursor := *stackCursor*′
        target := *target*′
    passRecStack := *passRecStack*′

SETRECUNDEF ≡
  exc := *undef*
  pass := *undef*
  stackCursor := *undef*
  target := *undef*
  handler := *undef*

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 `.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, this detail is 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 `.cctor` is embedded into

a `catch` handler. This `catch` handler catches exceptions of type `Object`, i.e. any exception, occured in `.cctor`, discards it, creates an object of type `TypeInitializationException`[8] and throws the new exception.

7 THE *Leave* PASS

The EXCCLR machine gets into the *Leave* pass when EXECCLR$_E$ 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 the *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.

$$
\begin{array}{l}
isFinFromTo(h, pos_1, pos_2) \Leftrightarrow \\
\quad isInTry(pos_1, h) \wedge clauseKind(h) = \texttt{finally} \wedge \\
\quad \neg isInTry(pos_2, h) \wedge \neg isInHandler(pos_2, h) \\
\\
isRealHanFromTo(h, pos_1, pos_2) \Leftrightarrow \\
\quad clauseKind(h) \in \{\texttt{catch}, \texttt{filter}\} \wedge \\
\quad isInHandler(pos_1, h) \wedge \neg isInHandler(pos_2, h)
\end{array}
$$

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 EXECCLR$_E$ execution (see Fig. 3).

[8] In the real CLR implementation, the exception thrown in `.cctor` is embedded as an inner exception in the `TypeInitializationException`. We do not model this aspect here.

8 THE RULES OF EXECCLR$_E$

The rules of EXECCLR$_E$ 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* $\neq$ *undef* then the current pass record is pushed on *passRecStack*.

$$
\begin{array}{l}
\textsc{LoadRec}(r) \equiv \\
\quad \textbf{if } r \in ExcPass \textbf{ then} \\
\quad\quad \textbf{let } (exc', pass', stackCursor', handler') = r \textbf{ in} \\
\quad\quad\quad exc := exc' \\
\quad\quad\quad pass := pass' \\
\quad\quad\quad stackCursor := stackCursor' \\
\quad\quad\quad handler := handler' \\
\quad \textbf{else let } (pass', stackCursor', target') = r \textbf{ in} \\
\quad\quad pass := pass' \\
\quad\quad stackCursor := stackCursor' \\
\quad\quad target := target' \\
\quad \textbf{if } pass \neq undef \textbf{ then } \textsc{PushRec} \\
\\
\textsc{PushRec} \equiv \\
\quad \textbf{if } pass = Leave \textbf{ then} \\
\quad\quad push(passRecStack, (pass, stackCursor, target)) \\
\quad \textbf{else } push(passRecStack, \\
\quad\quad\quad\quad (exc, pass, stackCursor, handler))
\end{array}
$$

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[9]:

$$
\begin{array}{l}
\textsc{Raise}(c) \equiv \\
\quad NewObj(c :: \texttt{.ctor}) \\
\quad Throw
\end{array}
$$

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

[9] The *NewObj* instruction called with an instance constructor `c::.ctor` creates a new object of class `c` and then calls the constructor `.ctor`.

Fig. 4 The rules of EXECCLR$_E$

```
EXECCLR_E(instr) ≡
  EXECCLR_N(instr)
  match instr
    Throw → let r = top(evalStack) in
              if r ≠ null then
                LOADREC((r, StackWalk, (frame, 0), undef))
                switch := ExcMech
              else RAISE(NullReferenceException)

    Rethrow → LOADREC((exc, StackWalk, (frame, 0), undef))
              switch := ExcMech

    EndFilter → let val = top(evalStack) in
                  if val = 1 then
                    FOUNDHANDLER
                    RESET(stackCursor, top(frameStack))
                  else GOTONXTHAN
                  POPFRAME
                  switch := ExcMech

    EndFinally → evalStack := []
                 switch := ExcMech

    Leave(pos) → evalStack := []
                 LOADREC((Leave, (frame, 0), pos))
                 switch := ExcMech
```

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,§12.4.2.5] which states *Execution cannot be resumed at the location of the exception, except with a user-filtered handler* and [1, Partition III,§3.34] 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. Independently of *val*, the current frame is discarded to reestablish the context of the faulting frame. Note that we do not explicitly pop *val* from the *evalStack* since 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[10] 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 for the mathematical correctness proof 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

[1] Common Language Infrastructure, Standard ECMA–335. `http://www.ecma-international.org/`. 2002.

[2] Chris Brumme. The Exception Model. Blog at `http://blogs.msdn.com/cbrumme/`, 2003.

[3] Matt Pietrek. A Crash Course on the Depths of Win32™ Structured Exception Handling. Microsoft Systems Journal, January 1997.

[4] Andrew D. Gordon and Don Syme. Typing a Multi-Language Intermediate Code. Technical Report Microsoft, MSR-TR-2000-106, December 2000.

[5] N. G. Fruja. Experiments with CLR. Example programs to determine the meaning of CLR features not specified by the ECMA standard. Available

[10] One can formally prove that there is such a "step" in the further run of the EXCCLR.

at http://www.inf.ethz.ch/personal/fruja/publications/clrexctests.pdf

[6] N. G. Fruja. Type Safety in C♯ and .NET CLR. PhD Thesis in preparation.

[7] N. G. Fruja. A Modular Design for the .NET CLR Architecture. Proceedings of the Workshop on Abstract State Machines, *ASM'05*, France.

[8] E. Börger, N. G. Fruja, V. Gervasi, R. F. Stärk. A High–Level Modular Definition of the Semantics of C♯. Journal Theoretical Computer Science, June, 2005.

[9] R. F. Stärk and E. Börger. An ASM Specification of C# Threads and the .NET memory model. Proceedings of the Workshop on Abstract State Machines, *ASM'04*, Germany, Springer LNCS 3052 (2004) pag. 38–60.

[10] E. Börger and R. F. Stärk. Exploiting Abstraction for Specification Reuse: The Java/C# Case Study. Formal Methods for Components and Objects: Second International Symposium, *FMCO'03*, The Netherlands, Springer LNCS 3188 (2004), pag. 42–76.

[11] N. G. Fruja. Specification and Implementation Problems for C♯. Proceedings of the Workshop on Abstract State Machines *ASM'04*, Germany, Springer LNCS 3052, pag. 127–143.

[12] N. G. Fruja. The Correctness of the Definite Assignment Analysis in C♯. Journal of Object Technology, vol. 3, no. 9, 2004.

[13] H. V. Jula and N.G. Fruja. An Executable Specification of C♯. Proceedings of the Workshop on Abstract State Machines, *ASM'05*, France.

[14] C. Marrocco. An Executable Specification of the .NET CLR. Diploma Thesis supervised by N. G. Fruja, ETH Zürich, 2005.

[15] R. F. Stärk, J. Schmid, E. Börger. Java and the Java Virtual Machine–Definition, Verification, Validation. Springer–Verlag, 2001.

[16] E. Börger and W. Schulte. A Practical Method for Specification and Analysis of Exception Handling – a Java JVM Case Study. *IEEE Transactions of Software Engineering*, vol. 26, 2000.

[17] E. Börger and R. F. Stärk. Abstract State Machines–A Method for High-Level System Design and Analysis. Springer-Verlag, 2003.

[18] Abstract State Machine Language (AsmL), Foundations of Software Engineering Group, Microsoft Research, Web pages at http://research.microsoft.com/foundations/AsmL/.