JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 2, No. 2, March–Aprile 2006

Modeling the .NET CLR Exception Handling Mechanism for a Mathematical Analysis

Nicu G. Fruja, Computer Science Department, ETH Zürich, Switzerland
Egon Börger, Dipartimento di Informatica, Università di Pisa, Italy

We provide a mathematical reference model for the exception handling mechanism of the Common Language Runtime (CLR), the virtual machine underlying the interpretation of .NET programs. The model fills some gap in the ECMA standard for CLR and is used to sketch the exception handling related part of a soundness proof for the CLR bytecode verifier.

1 INTRODUCTION

This work is part of a larger project [17] which aims at establishing some important properties of C♯ and CLR by mathematical proofs. Examples are the correctness of the bytecode verifier of CLR [11], the type safety (along the lines of the first author's correctness proof [14, 15] for the definite assignment rules) of C♯, the correctness of a general compilation scheme. We reuse the method developed for similar work for Java and the Java Virtual Machine (JVM) in [25]. As part of this effort, in [5, 13, 20] an abstract interpreter has been developed for C♯, including a thread and memory model [24, 23]; see also [8] for a comparative view of the abstract interpreters for Java and for C♯.

In [16] 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 [10]. Already in JVM the most difficult part for the correctness proof of the bytecode verifier was the one dealing with exception handling (see [25, §16]). The concrete purposes we are pursuing in this paper are twofold. First, we want to define a rigorous ground model (in the sense of [3]) for the CLR exception mechanism, to be used as reference model for the exception handling related part of a correctness proof for the bytecode verifier [11]. Secondly, we want to clarify the numerous issues concerning

Cite this document as follows: Nicu G. Fruja and Egon Börger: "Modeling the .NET CLR Exception Handling Mechanism for a Mathematical Analysis, in *Journal of Object Technology*, vol. , no. , April 2006, Special issue: .NET Technologies 2005 Conference, pp. -, http://www.jot.fm/issues/issue_2006_05/article

exception handling which are left open in the ECMA standard, but are relevant for a correct understanding of the CLR mechanism.

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 the few existing documents on the CLR exception handling [9]. The CLR mechanism has its origins in the Windows NT Structured Exception Handling (SEH) which is described in [22]. What we are striving for, the CLR type safety, is proved for a subset of CLR in [19]. However, that paper does not consider the exception handling classified in [19, §4] as *a fairly elaborate model that permits a unified view of exceptions in* C++, C♯*, and other high-level languages.* So far, no mathematical model has been developed for the CLR exception handling. The JVM exception mechanism, which differs significantly from the one of CLR, has been investigated in [6, 25].

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, the first author made a series of experiments with CLR, some of which are made available in [12] 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 [1]. The AsmL implementation of the entire CLR model is available in [21].

Since the focus of this paper is the exception mechanism of CLR, we assume the reader to have a rough understanding of CLR. We also refer without further explanations to the model EXECCLR$_N$ defined in [16], 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 [7]. However, for the reader's convenience we summarize here the most important ASM concepts and notations that are used in this paper. A state of an ASM is given by a set of static or 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

We stress the fact that in one step, an ASM fires simultaneously all its rules (synchronous parallelism).

Notational convention Beside the usual list operations (e.g., *push*, *pop*, *top*, *length*, and $\cdot$)[1], we use *split*(*L*,*1*) to split off the last element of the list L, i.e., *split*(*L*,*1*) is the pair $(L', [x])$ where $L' \cdot [x] = L$.

The paper is organized as follows. In Section 2, we list a few notations defined in [16] and used in this 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 considers the refinements that shall be applied to our model in order to also treat the handling of the special `ThreadAbortException`. In Section 10 we illustrate a verification usage of the mathematical CLR_E model by providing the exception handling related details of a soundness proof of (a model of) the CLR bytecode verifier. A preliminary version of this paper appeared in [18].

2 PRELIMINARIES

We summarize briefly the notations introduced in [16] that are relevant for the exception handling mechanism. For a detailed description we refer the reader to [16].

A method 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.

$$\text{PUSHFRAME} \equiv push(frameStack, frame)$$

[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 [16] 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** $\mathrm{EXECCLR}_E(code(pc))$

POPFRAME $\equiv$ **let** $(frameStack', [(pc', locAdr', argAdr', evalStack', meth')]) =$
$split(frameStack, 1)$ **in**
$pc := pc'$
$locAdr := locAdr'$
$argAdr := argAdr'$
$evalStack := evalStack'$
$meth := meth'$
$frameStack := frameStack'$

3 THE GLOBAL MACHINE STRUCTURE

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 [13]; this explains the definition of CLR_E as a machine which, in the normal execution mode (see also the remark below) 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 runtime 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

[3] The ECMA standard states in [10, 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.

- 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. The `filter` handlers bring considerable complexity to the exception 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:

> **Backstop entry** The *excHA* of the `entrypoint` method has as last entry a so-called *backstop entry* placed by the operating system which can handle any exception.

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.

Class initialization vs. Exception mechanism Although the ECMA standard [10, §8.9.5] says that a `beforefieldinit` class can be initialized at any time (before an access to one if its static fields occurs), it is not clear whether the .NET implementation follows the same line and allows such initialization to happen, for example, even during the purely administrative handler search EXCCLR has to accomplish to provide the specified effect of exception handling code, formally when $switch = ExcMech$. As one can see in Fig. 1, our model rules this out and considers that no initialization can happen when *switch* is *ExcMech*. This does not exclude initializations to be triggered during the execution of `filter` or handler code (when *switch* is different from *ExcMech*).

However, our model can be refined to allow class initializers to be non-deterministically triggered when *switch* is *ExcMech*:

- A stack *switchStack* of *switch* values is added.
- Assume that *switch* is *ExcMech* and the run-time system decides to initialize a `beforefieldinit` class. In this case, the current value of *switch*, i.e., *ExcMech*, is pushed onto *switchStack*, and the macro INITIALIZECLASS is executed.

[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.

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

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

- The rule for `Return` instructions is refined to reflect the special semantics of a *Return* instruction of a `.cctor` of a `beforefieldinit` class: *frame* is discarded, and *switch* is set to the topmost value on *switchStack*.

4 THE GLOBAL STRUCTURE 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.

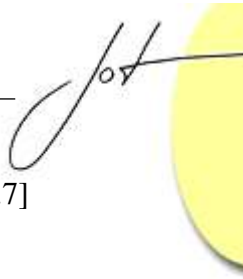
The elements of an exception handling array $\mathit{excHA} : \mathit{Map}(\mathit{MRef}, \mathit{List}(\mathit{Exc}))$ are known as *handlers* and can be of four kinds. They are elements of a set *Exc*:

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

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* is required by the ECMA standard to immediately follow the `filter` region – in particular we have $\mathit{filterStart} < \mathit{handlerStart}$. We often refer to the sequence of instructions between *filterStart* and $\mathit{handlerStart} - 1$ 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 presentation, we define the predicates in Fig. 2 for an instruction located at program counter position $\mathit{pos} \in \mathit{Pc}$ and a handler $h \in \mathit{Exc}$. Note that if the predicate *isInFilter* is true, then *filterStart* is defined and therefore *h* is of kind `filter`. Based on

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



the lexical nesting constraints of protected blocks specified in [10, Partition I,§12.4.2.7] which we assume in the model, one can prove the following property:

Disjointness 1 *The predicates isInTry, isInHandler and isInFilter are pairwise disjoint.*

We also assume all the constraints concerning the lexical nesting of handlers specified in the standard [10, Partition I,§12.4.2.7]. The ECMA standard [10, Partition I,§12.4.2.5] ordering assumption on handlers is:

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

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 f) and in the exception handling array of f (an index in *excHA*),
- the suitable *handler* determined at the end of the *StackWalk* pass (if any); this 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 [10, §12.4.2.8, Partition I], every normal execution of a `try` block or a `catch`/`filter` handler region (not to be confused with a `filter` region) must end with a $Leave(target)$ 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.

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

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

Leave constraints:

1. It is not legal to exit with a *Leave* instruction a `filter` region or 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.
4. A *Leave* instruction is executed *only* upon the normal exit from a `try` block or a `catch`/`filter` handler region.
5. The target of any branch instruction, in particular of $Leave(target)$, points to an instruction within the same method as the branch instruction.

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

$$passRecStack : List(ExcRec \cup LeaveRec) \qquad passRecStack = [\,]$$

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

Only one handler region per `try` block? The ECMA standard specifies in [10, 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:

$$\begin{array}{l} \texttt{.try}\ \{ \\ \quad block \\ \}\ \texttt{catch}\ block_1 \\ \quad \texttt{catch}\ block_2 \end{array} \quad \Longrightarrow \quad \begin{array}{l} \texttt{.try}\ \{ \\ \quad \texttt{.try}\ \{ \\ \qquad block \\ \quad \}\ \texttt{catch}\ block_1 \\ \}\ \texttt{catch}\ block_2 \end{array}$$

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 may exist if the search position n in the current frame has not yet reached the last element of the array *excHA* of handlers of the corresponding method m.

$$existsHanWithinFrame((\_,\_,\_,\_,m),n) \Leftrightarrow n < length(excHA(m))$$

If there are no (more) handlers in the frame fr pointed to by *stackCursor*, then the search has to be continued at the invoker frame fr'. This means to reset the *stackCursor* to point to the invoker frame, which precedes fr in the frame stack combined with *frame*:

$$\text{SEARCHINVFRAME}(fr) \equiv \textbf{let } \_ \cdot [fr',fr] \cdot \_ = frameStack \cdot [frame] \textbf{ in } \text{RESET}(stackCursor, fr')$$

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;

$$matchCatch(pos,t,h) \Leftrightarrow isInTry(pos,h) \wedge clauseKind(h) = \texttt{catch} \wedge t \preceq type(h)$$

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

$$matchFilter(pos,h) \Leftrightarrow isInTry(pos,h) \wedge clauseKind(h) = \texttt{filter}$$

- a `filter` handler whose `filter` region contains the *pc* of the frame pointed at by *stackCursor*. This corresponds to an outer exception described below.

The order of the **if** clauses in the **let** statement from the rule *StackWalk* in Fig. 3 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*[6].

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

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

The handler pointed to by the *stackCursor*, namely $hanWithinFrame((\_,\_,\_,\_,m),n)$, is defined to be $excHA(m)(n)$. If this handler is not of one of the three types above, then the *stackCursor* is incremented to point to the next handler candidate in the *excHA*:

$$\textsc{GoToNxtHan} \equiv stackCursor := (fr, n+1) \quad \textbf{where } stackCursor = (fr, n)$$

The *Ordering assumption* stated in Section 4 and the lexical nesting constraints stated in [10, Partition I,§12.4.2.7] ensure that if the *stackCursor* points to a handler of one of the three types above, then this handler is the first such handler in the exception handling array (starting at the position indicated in the *stackCursor*).

Handler Case 1 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.

$$\begin{array}{l}\textsc{FoundHandler} \equiv \\ \quad pass := Unwind \\ \quad handler := stackCursor \\ \\ \textsc{Reset}(s,fr) \equiv s := (fr, 0)\end{array}$$

Handler Case 2 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 detail is omitted by the ECMA standard [10]. 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 $\textsc{execCLR}_E$.

[7] We use the *actualTypeOf* function defined in [16] to determine the runtime type of the exception.

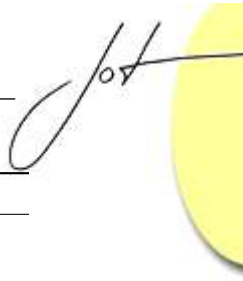


Fig. 3 The exception handling machine EXCCLR

```
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(fr)
              where stackCursor = (fr, _) and fr = (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
            evalStack := [ ]
            POPREC
            switch := Noswitch
```

```
EXECFILTER(h) ≡ pc       := filterStart(h)
                evalStack := [exc]
                locAdr    := locAdr′
                argAdr    := argAdr′
                meth      := meth′
                PUSHFRAME
                switch := Noswitch
                where stackCursor = ((_, locAdr′, argAdr′, _, meth′), _)
```

Handler Case 3 The *stackCursor* points to a `filter` handler whose `filter` region contains the *pc* of the frame pointed at by *stackCursor*.

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 inner 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 inner exception is not taken care of in the `filter` region, then it will be discarded (via the CONTINUEOUTEREXC macro defined in Section 6) after its `finally` and `fault` handlers have been executed (see `Tests` 6, 8, and 9 in [12]). Therefore, in this case EXCCLR exits via the macro EXITINNEREXC the *StackWalk* and starts an *Unwind* pass, during which all the `finally`/`fault` handlers for the inner exception are executed until the `filter` region where the inner exception occured is reached.

EXITINNEREXC $\equiv$
  $pass := Unwind$
  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 one of the following four handlers:

- 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 an exception was thrown in a `filter` region. For the matching target handler case, 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.

$$matchTargetHan((fr', n'), (fr'', n'')) \Leftrightarrow sameFrame(fr', fr'') \wedge n' = n''$$

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

- 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 of the rule *Unwind* in Fig. 3 is not important. It only matters that the first clause is guarded by *matchTargetHan*.

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

The property follows from the definitions and the property *Disjointness* 1.

The *Ordering assumption* in Section 4 and the lexical nesting constraints given in [10, 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 not of any of the above four types, the *stackCursor* is incremented to point to the next handler in the *excHA*.

Handler Case 1 The handler pointed to by the *stackCursor* is the *handler* found in the *StackWalk*. Then the handler region of *handler* 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, nothing is pushed onto *evalStack*), and the control switches to EXECCLR$_E$.

$$\begin{array}{l}\textsc{ExecHan}(h) \equiv \\ \quad pc := handlerStart(h) \\ \quad evalStack := \textbf{if } clauseKind(h) \in \{\texttt{catch}, \texttt{filter}\} \textbf{ then } [exc] \\ \qquad\qquad\qquad \textbf{else } [\,] \\ \quad switch := Noswitch\end{array}$$

Handler Case 2 The handler pointed to by the *stackCursor* is a matching `finally` or `fault` handler. Then its handler region is executed with initially empty evaluation stack. At the same time, the *stackCursor* is incremented through GOTONXTHAN.

Handler Case 3 The handler region of the handler pointed to by *stackCursor* 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 [12] led to the following rules of thumb for exceptions thrown in a handler region, in a way similar 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 following two actions are taken:

  – the previous pass of EXCCLR is aborted through ABORTPREVPASSREC;

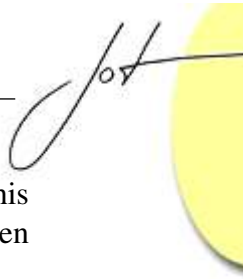
  $$\text{ABORTPREVPASSREC} \equiv pop(passRecStack)$$

  – the exception is propagated further via GOTONXTHAN in the *Unwind* pass which sets the *stackCursor* to the next handler in *excHA*.

This implies that an exception can go "unhandled" without taking down the process, namely if an outer exception goes unhandled, but an inner exception is successfully handled. In fact, the execution of a handler region can only occur when EXCCLR runs in the *Unwind* or *Leave* pass: 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 was running 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 [12]). 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 [12]).

Handler Case 4 The handler pointed to by the *stackCursor* is a `filter` handler whose `filter` region contains *pc*. Then the execution of this `filter` region must have triggered an inner exception whose *StackWalk* led to a call of EXITINNEREXC. In this case, the current (inner) exception is aborted, and the `filter` considered as not providing a handler for the outer exception. Formally, CONTINUEOUTEREXC 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 the macro POPREC. Note that by these stipulations, 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 execution of POPFRAME is safe since the *frameStack* cannot be empty at the time when CONTINUEOUTEREXC is fired. [10, Partition II, §12.4.2.8.1] states that the control can be transferred to a `filter` region only through EXCCLR. Since *pc* is in



a `filter` region, an EXECFILTER should have been already executed. But in this case, a new frame is pushed on the *frameStack*. Hence, *frameStack* is not empty when CONTINUEOUTEREXC is executed.

$$
\begin{array}{l}
\textsc{ContinueOuterExc} \equiv \\
\quad \textsc{PopFrame} \\
\quad \textsc{PopRec}\ \textbf{seq}\ \textsc{GoToNxtHan}
\end{array}
$$

$$
\begin{array}{l}
\textsc{PopRec} \equiv \textbf{if}\ passRecStack = [\,]\ \textbf{then} \\
\qquad\qquad \textsc{SetRecUndef} \\
\qquad\qquad switch := Noswitch \\
\qquad\quad \textbf{else let}\ (passRecStack', [r]) = split(passRecStack, 1)\ \textbf{in} \\
\qquad\qquad \textbf{if}\ r \in ExcRec\ \textbf{then} \\
\qquad\qquad\quad \textbf{let}\ (exc', pass', stackCursor', handler') = r\ \textbf{in} \\
\qquad\qquad\qquad exc := exc' \\
\qquad\qquad\qquad pass := pass' \\
\qquad\qquad\qquad stackCursor := stackCursor' \\
\qquad\qquad\qquad handler := handler' \\
\qquad\qquad \textbf{if}\ r \in LeaveRec\ \textbf{then} \\
\qquad\qquad\quad \textbf{let}\ (pass', stackCursor', target') = r\ \textbf{in} \\
\qquad\qquad\qquad pass := pass' \\
\qquad\qquad\qquad stackCursor := stackCursor' \\
\qquad\qquad\qquad target := target' \\
\qquad\qquad passRecStack := passRecStack'
\end{array}
$$

$$
\begin{array}{l}
\textsc{SetRecUndef} \equiv exc := undef \\
\qquad\qquad\qquad\quad pass := undef \\
\qquad\qquad\qquad\quad stackCursor := undef \\
\qquad\qquad\qquad\quad target := undef \\
\qquad\qquad\qquad\quad handler := undef
\end{array}
$$

If the *Unwind* pass exhausted all the handlers in the frame indicated in *stackCursor*, the current frame is popped from *frameStack* and the *Unwind* pass continues in the invoker frame of the current frame. Note that the execution in the **else** clause of the macros POPFRAME and SEARCHINVFRAME is safe as *frame* has a caller frame, i.e., *frame* cannot be the frame of the `entrypoint`. This is because *Backstop entry* guarantees that the **else** clause is not reachable if *frame* is the frame of the `entrypoint`. The same argument can be invoked also in case of SEARCHINVFRAME in the *StackWalk* pass.

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 the specific exception `TypeInitializationException` is thrown. This means that an exception can

and will escape the body of a `.cctor` only by a `TypeInitializationException`. Any further attempt to access the corresponding class in the current application domain will throw *the same* `TypeInitializationException` object. 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 $C^\sharp$ in the ECMA standard (see `Test` 7 in [12]). 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, which by the *Leave* constraints can happen only upon a 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 frame of the *Leave* instruction (see Fig. 4). 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.

Handler Case 1 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 first *Leave* rule in Fig. 3).

Handler Case 2 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 second *Leave* rule in Fig. 3). In fact, the discarded record refers to the *Unwind* pass for handling an exception by executing the `catch`/`filter` handler pointed at by *stackCursor*, thus terminating the handling of the corresponding exception.

$$\begin{aligned} &\mathit{isFinFromTo}(h, pos', pos'') \Leftrightarrow \mathit{clauseKind}(h) = \texttt{finally} \wedge \mathit{isInTry}(pos', h) \wedge \\ &\qquad\qquad \neg\mathit{isInTry}(pos'', h) \wedge \neg\mathit{isInHandler}(pos'', h) \\ &\mathit{isRealHanFromTo}(h, pos', pos'') \Leftrightarrow \mathit{clauseKind}(h) \in \{\texttt{catch}, \texttt{filter}\} \wedge \\ &\qquad\qquad \mathit{isInHandler}(pos', h) \wedge \neg\mathit{isInHandler}(pos'', h) \end{aligned}$$

[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.

Although the two **if** clauses in the **let** statement from the *Leave* pass are executed in parallel, it is never the case that the embedded EXECHAN and ABORTPREVPASSREC are simultaneously executed. The reason is given by the following property which can be easily proved using the definitions of the predicates:

Disjointness 4 *The predicates isFinFromTo and isRealHanFromTo are disjoint.*

For each handler EXCCLR inspects also the next handler in *excHA*. When the handlers in the current method are exhausted, by the *Leave* constraints this round of EXCCLR is terminated, and the execution proceeds at *target*: *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 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 the pass record associated to the given exception: the *stackCursor* is initialized for a *StackWalk* 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 onto *passRecStack*.

```
LOADREC(r) ≡
  if r ∈ ExcPass then
    let (exc′, pass′, stackCursor′, handler′) = r in
      exc         := exc′
      pass        := pass′
      stackCursor := stackCursor′
      handler     := handler′
  else let (pass′, stackCursor′, target′) = r in
    pass        := pass′
    stackCursor := stackCursor′
    target      := target′
  if pass ≠ undef then PUSHREC

PUSHREC ≡
  if pass = Leave then push(passRecStack, (pass, stackCursor, target))
  else push(passRecStack, (exc, pass, stackCursor, handler))
```

If the exception reference found on top of the *evalStack* by the *Throw* instruction is `null`, a `NullReferenceException` is thrown. For a given class c, the macro RAISE(c) is

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     → switch := ExcMech

    Leave(target)  → LOADREC((Leave, (frame, 0), target))
                     switch := ExcMech
```

defined by the following code template[9]:

```
RAISE(c) ≡ NewObj(c :: .ctor)
           Throw
```

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 [10, Partition III,§4.23] that the *Rethrow* instruction is only permitted within the body of a `catch` handler. However, in reality it is allowed also within a handler region of a `filter` (see `Test` 5 in [12]) throwing 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, exactly one *EndFilter* is allowed, namely its last instruction, which is supposed to be the only one used to normally exit the `filter` region (see the

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

remark above on exceptions in a 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 [10, Partition I,§12.4.2.5] which states *Execution cannot be resumed at the location of the exception, except with a user-filtered handler* – therefore a "resume exception" value in addition to 0 and 1 is foreseen allowing CLR to resume the execution at the point where the handled exception has been raised— and [10, 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).

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 (normally) the execution of the handler region of a `finally` or of a `fault` handler. It transfers the control to EXCCLR. A *Leave* instruction loads 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 *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 will propagate 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 (see Fig. 5) and is not ensured by the exception handling mechanism.

9 THE THREADABORTEXCEPTION

There is one exception, i.e., `ThreadAbortException` [2], whose handling needs an extension of our exception model. When a call is made to `Thread`::`Abort` to terminate a thread, the system throws a `ThreadAbortException` in the *target thread*. `ThreadAbortException` is a special exception (known also as an "unstoppable" exception) that can be caught by application code, but is rethrown at the end of the `catch`/`filter` handler region unless the method `Thread`::`ResetAbort` is called. When the `ThreadAbortException` is raised, the exception mechanism executes any

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

`finally`/`fault` handler regions for the target thread.

As the ECMA standard [10] does not specify the special handling of this exception, we did not include it in our basic model. However, the model is flexible enough to be refined in a few places (in the sense of ASM *refinements* defined in [4]) to also cover the handling of this "unstoppable" exception:

- The universe *ExcRec* of exception records is refined to include an object reference denoted *discardedTAE*.

$$
\begin{array}{lcll}
ExcRec & = & ExcRec\,(& exc : ObjRef \\
 & & & pass : \{StackWalk, Unwind\} \\
 & & & stackCursor : Frame \times \mathbb{N} \\
 & & & handler : Frame \times \mathbb{N} \\
 & & & discardedTAE : ObjRef\ )
\end{array}
$$

  Assume that a `ThreadAbortException` is handled by the exception handling mechanism EXCCLR. If another exception, say *exc*, is raised, and the handling of *exc* attempts to discard the `ThreadAbortException`, then the discarded `ThreadAbortException` reference is stored in *discardedTAE*.

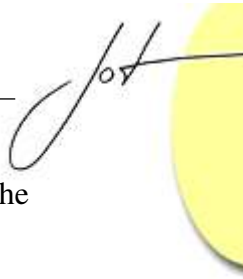
  Let us assume that the current thread is going to be aborted. We assume that an exception record associated to a `ThreadAbortException` is loaded into EXCCLR. The *discardedTAE* component of the record is set to the exception reference. Thus, the components *exc* and *discardedTAE* are the same.

- The macro ABORTPREVPASSREC used in the *isInHandler* clause of the *Unwind* rule is refined to also "transfer" the abort request (if any), i.e., the current exception will take over the `ThreadAbortException` reference (if any) carried by the discarded exception record:

$$
\begin{array}{l}
\textsc{AbortPrevPassRec} \equiv \\
\quad pop(passRecStack) \\
\quad \textbf{let } r = top(passRecStack) \textbf{ in} \\
\quad\quad \textbf{if } r \in ExcRec \textbf{ then} \\
\quad\quad\quad \textbf{let } (\_, \_, \_, \_, discardedTAE') = r \textbf{ in} \\
\quad\quad\quad\quad \textbf{if } discardedTAE' \neq undefined \textbf{ then} \\
\quad\quad\quad\quad\quad discardedTAE := discardedTAE'
\end{array}
$$

  Similarly, POPREC used in the *isInFilter* clause of the *Unwind* rule is refined to "transfer" the `ThreadAbortException` that has to be raised later again. Note that in this way a `ThreadAbortException` can escape a `filter` region.

  Also, the macro SETRECUNDEF is refined to also reset *discardedTAE* to *undefined*.



- The *isRealHanFromTo* clause of the *Leave* rule in Fig. 3 is modified to rethrow the discarded `ThreadAbortException`:

$$
\begin{array}{l}
\textbf{let } r = top(passRecStack) \textbf{ in} \\
\quad \textbf{let } (\_,\_,\_,\_,discardedTAE') = r \textbf{ in} \\
\quad\quad \textbf{if } discardedTAE' \neq undefined \textbf{ then} \\
\quad\quad\quad \textsc{PopRec} \textbf{ seq} \\
\quad\quad\quad\quad exc := discardedTAE' \\
\quad\quad\quad\quad pass := StackWalk \\
\quad\quad\quad\quad stackCursor := (frame, 0) \\
\quad\quad\quad\quad handler := undefined \\
\quad\quad\quad\quad discardedTAE := discardedTAE' \\
\quad\quad \textbf{else } pop(passRecStack)
\end{array}
$$

  Note that the incrementation of the *stackCursor* through GOTONXTHAN shall not be anymore done in the *isRealHanFromTo* clause but only for the *isFinFromTo* clause.

- The special semantics of invoking the `Thread`::`ResetAbort` method has to be added to the definition of the machine CLR_E in Fig. 4. Beside executing the method body, the invocation also *aborts* the `ThreadAbortException`. Note that the *abort* does not stop the handling of the `ThreadAbortException` but only its "unstoppable" attribute. In other words, after the `ThreadAbortException` is handled by EXCCLR, the execution continues normally and the exception is not raised automatically again.

  The *abort* is realized by setting to *undefined* the *discardedTAE* component of the exception records on *passRecStack*[11].

10 THE BYTECODE VERIFICATION

The bytecode verifier statically checks the type-safety of the bytecode and therefore its soundness is critical for the security model. We show in this section how one can use the mathematical model introduced in the previous sections in the soundness proof of the .NET CLR bytecode verifier specified in [10, Partition III]. We provide arguments to establish the soundness of the bytecode verification in case exception handling related steps could be performed by the code, assuming the soundness for the verification of code related only to exception free ($\mathrm{EXECCLR}_N$) execution. More precisely, we sketch a proof that, for methods accepted by the verifier, the execution of instructions related to exceptions does not violate any of the following properties: *type safety*, i.e., every instruction will always execute with arguments of expected types, *bounded evaluation*

[11] One can formally prove that, at a given time, there exists at most one exception record that has the *discardedTAE* component defined.

Fig. 5 Verifying the instructions related to exceptions

$$
\begin{array}{l}
\mathit{check}(\mathit{meth}, \mathit{pos}, \mathit{evalStackT}) \Leftrightarrow \textbf{match}\ \mathit{code}(\mathit{pos}) \\
\qquad\qquad \begin{array}{ll}
\mathit{Throw} & \rightarrow\ \mathit{top}(\mathit{evalStackT}) \sqsubseteq \texttt{object} \\
\mathit{Rethrow} & \rightarrow\ \mathit{True} \\
\mathit{EndFilter} & \rightarrow\ \mathit{evalStackT} = [\texttt{int32}] \\
\mathit{EndFinally} & \rightarrow\ \mathit{True} \\
\mathit{Leave}(\_) & \rightarrow\ \mathit{True}
\end{array}
\end{array}
$$

stack, i.e., the evaluation stack will never exceed a bound computed by the compiler, *program counter safety*, i.e., the program counter will always point to a valid code index.

What the verifier checks

For the soundness proof, we do not rely upon any particular bytecode verifier but list only the assumptions we need on the execution of the bytecode verification as described in [10, Partition III]. The bytecode verification is performed on a per-method basis. The verifier simulates all possible control flow paths through the code, attempting to associate a valid *type stack state*[12] with every reachable instruction. The type stack state *evalStackT* specifies for each slot of *evalStack* a required type in that slot and thereby also the number of values on the *evalStack* at that point in the code. Before simulating the execution of an instruction, the verifier checks whether certain conditions are satisfied. We specify in Fig. 5 by means of the predicate *check* the conditions checked by the verifier for the instructions related to exceptions. The checks for a single instruction operate on *evalStackT*. The relation $\sqsubseteq$ denotes the *compatibility relation* between types; for a formal definition see [11].

In JVM, the *Throw* instruction expects an object of type `Throwable` on the stack. The CLR bytecode verifier is not so strict: it requires that the top stack element is of type `object`. The *EndFilter* instruction which terminates the execution of a `filter` region expects an integer on the stack and that the stack contains only this integer. For the instructions *Rethrow*, *EndFinally*, and *Leave* nothing has to be checked.

Since the bytecode verifier works on a stack of types and not of values, at branching points in the control-flow it has to consider every successor that may be possible at runtime. Therefore, the type stack state for an instruction at *pos* yields constraints on how to match the type stack states of all instructions that are runtime possible control-flow successors of *pos*. In Fig. 6, we define the function *succ* which, given an instruction and a type stack state, computes the successor code indices together with their type stack states.

Each instruction can throw exceptions. This assumption is realistic since the special `ExecutionEngineException` may be thrown at any time during the execution of a program. Therefore, for an instruction at *pos*, all the handlers *h* that protect *pos*

[12]The ECMA standard uses the name *stack state*. However, we prefer *type stack state* since we have a more complex notion of state and also more than one stack.

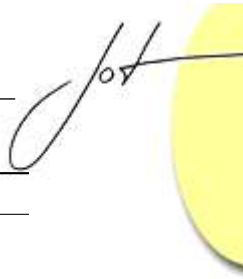


Fig. 6 The type stack state successors of the instructions related to exceptions

$succ(meth, pos, evalStackT) = excHandlers(meth, pos) \cup$
  match $code(pos)$
    $Throw \rightarrow \emptyset$
    $Rethrow \rightarrow \emptyset$
    $EndFilter \rightarrow \emptyset$
    $EndFinally \rightarrow \{(target, [\,]) \mid target \in LeaveThroughFin(meth, pos)\}$
    $Leave(target) \rightarrow$ **if** $\{h \in excHA(meth) \mid isFinFromTo(h, pos, target)\} = \emptyset$ **then**
      $\{(target, [\,])\}$
    else $\emptyset$

are included into the set of possible successor type stack states by means of a function *excHandlers*. Upon entering a `catch` handler, the type stack state contains only the type $type(h)$ of exceptions that h is "handling" whereas, upon entering a `filter` region or a `filter` handler region, the type stack state is [`object`]. In case of a `finally`/`fault` handler, the type stack state is $[\,]$. Except for the case of a `filter` region, the successor code index is given by $handlerStart(h)$. In case of a `filter` region, the successor is $filterStart(h)$.

$excHandlers(meth, pos) = \{\, (handlerStart(h), [type(h)]) \mid h \in excHA(meth)$
  and $isInTry(pos, h)$ and $clauseKind(h) =$ `catch`$\}\ \cup$
  $\{\, (filterStart(h), [$`object`$]), (handlerStart(h), [$`object`$])$
  $\mid h \in excHA(meth)$ and $isInTry(pos, h)$ and $clauseKind(h) =$ `filter`$\}\ \cup$
  $\{\, (handlerStart(h), [\,]) \mid h \in excHA(meth)$ and $isInTry(pos, h)$
  and $clauseKind(h) \in \{$`finally`, `fault`$\}\,\}$

Beside the successors defined by *excHandlers*, in case of the instructions *Throw*, *Rethrow*, and *EndFilter* there are no other successors (see Fig. 6). In case of an *EndFinally* instruction *succ* yields also the targets of *Leave* instructions that could trigger the execution of the `finally` handler to which the *EndFinally* instruction corresponds. The associated type stack state is the empty list. The set *LeaveThroughFin* of these possible targets is defined as follows.

$LeaveThroughFin(meth, pos) = \{\, target \in Pc \mid \exists\, pos' \in Pc$ such that
  $code(pos') = Leave(target)$ and for the h such that
  $[h' \in excHA(meth) \mid isFinFromTo(h', pos', target)] = [\ldots, h]$
  holds $[h' \in excHA(meth) \mid isInHandler(pos, h')] = [h, \ldots]\,\}$

Thus, *target* is an element of $LeaveThroughFin(meth, pos)$ for an *EndFinally* instruction at *pos* if this *EndFinally* corresponds to the last `finally` handler "on the way" from a

Leave(*target*) instruction to the instruction at *target*. By definition, the set *LeaveThroughFin* is empty for an *EndFinally* instruction which corresponds to a `fault` handler (because the predicate *isFinFromTo* evaluates to false for `fault` handlers). This means that the *EndFinally* instruction which terminates the execution of a `fault` handler region has no successors (beside those defined in *excHandlers*).

We now explain the definition of *succ* in the case of a *Leave* instruction. If there is no `finally` handler "on the way" from the *Leave* instruction to its target, the target instruction with an empty type stack state is an additional successor. If there is a `finally` handler "on the way", say h, the successor given by the instruction at *handlerStart*(h) is already considered in *excHandlers*, so that in this case *succ* provides no additional successor.

The context of the verifier soundness proof

Instead of a particular bytecode verifier, we use a characterization of the type properties of bytecode that is accepted by the verifier. This leads us to Definition 1 of well-typedness of a method, which we consider as a requirement for every method to be accepted by the verifier. The auxiliary relation $\sqsubseteq_{len}$ of pointwise compatibility of type stack states is defined for lists of types L', L'' of lengths m, n as follows:

$$L' \sqsubseteq_{len} L'' \quad \Leftrightarrow \quad m = n \text{ and } L'(i) \sqsubseteq L''(i) \text{ for every } i < m.$$

Definition 1 (Well-typed method) *A method mref is called* well-typed *if there exists a family of type stack states* $(evalStackT_i)_{i\in\mathcal{D}}$ *over a domain* $\mathcal{D}$ *which satisfies the conditions:*

(wt1) *The elements of* $\mathcal{D}$ *are valid code indices of mref.*

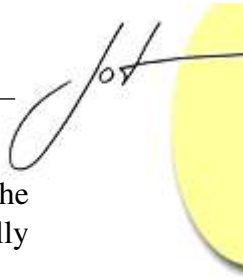
(wt2) $0 \in \mathcal{D}$.

(wt3) $evalStackT_0 = [\,]$.

(wt4) *If* $i \in \mathcal{D}$, *then* $check(mref, i, evalStackT_i)$ *is true.*

(wt5) *If* $i \in \mathcal{D}$ *and* $(j, evalStackT') \in succ(mref, i, evalStackT_i)$, *then* $j \in \mathcal{D}$ *and* $evalStackT' \sqsubseteq_{len} evalStackT_j$.

The domain $\mathcal{D}$ collects the code indices which are reachable from the code index 0. (**wt1**) states that $\mathcal{D}$ consists of valid code indices only, and (**wt2**) says that 0 is in the domain. (**wt3**) requires the type stack state to be empty upon the method entry. (**wt4**) ensures that the type stack states satisfy all type-consistency checks. (**wt5**) says that a successor type stack state has to be more specific than the type stack state associated to the successor index. In particular, this means that the type stack state asserted in any successor shall be more specific but of the same length as the predecessor type stack state. The condition is related to the rules described in [10, Partition III, §1.8.1.3] for "merging" type stack states. When simulating the control flow paths, the verifier merges the simulated type stack state with the existing type stack state at any successor instruction in the flow. If the numbers



of slots in the two type stack states differ, the merge fails. Otherwise, the merge of the type stack states is computed slot-by-slot. A precise abstract definition of the structurally similar JVM bytecode verifier formalizing such merge rules is provided in [25, §17].

The proof for CLR_E

In this section, we show how to extend the soundness proof from $\mathrm{EXECCLR}_N$ to CLR_E. The soundness theorem proved for the exception-free machine $\mathrm{EXECCLR}_N$ guarantees that the following type-safety invariants hold at runtime for well-typed methods.

(pc) $pc \in \mathcal{D}$, i.e., the program counter *pc* is always a valid code index;

(stack1) the current *evalStack* has the same length as $evalStackT_{pc}$;

(stack2) the values on the current *evalStack* are compatible with the corresponding types assigned in $evalStackT_{pc}$;

(loc) all the local variables have values compatible with the declared local variable types;

(arg) all the arguments have values compatible with the declared argument types;

(init) an "uninitialized" object can only occur in an object class constructor `.ctor` of an appropriate class (see the object initialization rules in [10, Partition III, §1.8.1.4]);

(field) all fields of an object class instance are compatible with the declared field types;

(box) the value type instance embedded into a boxed value (object) is of the expected value type;

As one can easily see in the proof of Theorem1, the invariants (**loc**)-(**box**) are not affected by computation steps of EXCCLR. It suffices therefore to consider here only the invariants (**pc**), (**stack1**), and (**stack2**). The formalization of the invariant (**stack2**) involves a typing judgment $\vdash val : t$ defined in [11] and interpreted as follows: the type of the value *val* is a subtype of the type *t*.

Theorem 1 (Soundness of Bytecode Verification) *The following invariants are satisfied by every frame in every runtime state of* CLR_E *for every well-typed method meth:*

(pc) $pc \in \mathcal{D}$

(stack1) $length(evalStack) = length(evalStackT_{pc})$

(stack2) $\vdash evalStack(j) : evalStackT_{pc}(j)$, *for every* $j < length(evalStack)$

Proof. The proof assumes the proof for EXECCLR$_N$ and proceeds by induction on the run of CLR$_E$. In the initial state of CLR$_E$, the invariants for the single existing frame (of the `.entrypoint` method) are satisfied. In fact, this frame satisfies $pc = 0$ and $evalStack = [\,]$. Thus, (**pc**) holds by (**wt2**) and (**stack1**), and (**stack2**) follows from (**wt3**). In the induction step, we proceed by case distinction on $code(pc)$, whether EXECCLR$_E$ has the control (*switch* = *Noswitch*) or EXCCLR (*switch* = *ExcMech*). In the second case, we make an additional case distinction on *pass*. Due to space limitations we present here only some characteristic subcases that can occur in each case.

Case 1 $switch = Noswitch$. The subcases $code(pc) \in \{Throw, Rethrow, EndFinally\}$ are trivial because they do not affect neither *pc* nor *evalStack*.

Subcase 1.1 $code(pc) = EndFilter$. Then CLR$_E$ executes POPFRAME, and sets *switch* to *Noswitch*. POPFRAME reestablishes the context of the invoker frame of *frame*. The induction hypothesis guarantees that the invariants hold for the new current frame, namely the caller frame of *frame*.

Subcase 1.2 $code(pc) = Leave(target)$. When executing $Leave(target)$, EXECCLR$_E$ loads the leave record $(Leave, (frame, 0), target)$ onto the *passRecStack* and gives the control to EXCCLR, i.e., sets *switch* to *ExcMech*. As the invariants are not affected, the claim follows from the induction hypothesis.

Case 2 $switch = ExcMech$. We present two subcases out of three.

Subcase 2.1 $pass = Leave$. Let *s* and *target* be the current value of the *stackCursor* and the target associated to this *Leave* pass, respectively.

Subcase 2.1.1 $existsHanWithinFrame(s)$ is true. Let *h* be the handler pointed to by *s*.

If $isRealHanFromTo(h, pc, target)$ is true, EXCCLR executes ABORTPREVPASSREC and GOTONXTHAN which do not influence the invariants. Therefore, the claim follows from the induction hypothesis.

If $isFinFromTo(h, pc, target)$ is true, EXCCLR executes the macros EXECHAN(h) and GOTONXTHAN. EXECHAN(h) sets *pc* to $handlerStart(h)$, *evalStack* to $[\,]$ (since by the definition of *isFinFromTo*, *h* is a `finally` handler) and *switch* to *Noswitch*. By the definition of *excHandlers*, we get that $(handlerStart(h), [\,])$ is in the set $excHandlers(meth, pc)$ and therefore in $succ(meth, pc, evalStackT_{pc})$. This together with (**pc**) and (**wt5**) imply $handlerStart(h) \in \mathcal{D}$ and $[\,] \sqsubseteq_{len} evalStackT_{handlerStart(h)}$. This means that the invariants (**pc**), (**stack1**), and (**stack2**) are preserved for the current frame (the last two invariants hold since $evalStackT_{handlerStart(h)}$ is necessarily $[\,]$).

Subcase 2.1.2 $existsHanWithinFrame(s)$ is false. In this case, EXCCLR sets *pc* to *target*, *evalStack* to $[\,]$, *switch* to *Noswitch* and executes POPREC. From the definition of the *Leave* rules in EXECCLR$_E$ and EXCCLR, it follows that $code(pc) = EndFinally$ or $code(pc) = Leave(target)$.

If $code(pc)$ is the *EndFinally* instruction, the definition of the function *succ* implies $(target, [\,]) \in succ(meth, pc, evalStackT_{pc})$.

If $code(pc)$ is an instruction $Leave(target)$, then by the definition of Subcase 2.1.2, the set $\{h \in excHA(meth) \mid isFinFromTo(h, pc, target)\}$ is empty. Then the definition of *succ* implies $(target, [\,]) \in succ(meth, pc, evalStackT_{pc})$.

Thus, in every case holds $(target, [\,]) \in succ(meth, pc, evalStackT_{pc})$. This together with (**pc**) and (**wt5**) implies $target \in \mathcal{D}$ and $[\,] \sqsubseteq_{len} evalStackT_{target}$. Consequently, the invariants (**pc**), (**stack1**), and (**stack2**) hold for the current frame (the last two hold since $evalStackT_{target}$ shall be $[\,]$).

Subcase 2.2 $pass = StackWalk$. Let s be the current value of the *stackCursor*.

Subcase 2.2.1 $existsHanWithinFrame(s)$ is true. Let h be the handler pointed to by s and pos the program counter of the frame pointed to by s.

If h satisfies $matchFilter(pos, h)$, then EXCCLR executes EXECFILTER(h) which loads on the *frameStack* a frame with *pc* set to $filterStart(h)$ and *evalStack* to $[exc]$. From the definition of *matchFilter*, we have $isInTry(pos, h)$ and $clauseKind(h) =$ `filter`. By this and the definitions of *succ* and *excHandlers*, we get $(filterStart(h), [\texttt{object}]) \in succ(mref, pos, evalStackT_{pos})$ where *mref* is the method of the frame pointed to by s. The definition of EXECFILTER implies that *mref* is also the method of the new current frame. Thus, $(filterStart(h), [\texttt{object}]) \in succ(meth, pos, evalStackT_{pos})$. This, (**pc**), and (**wt5**) imply $filterStart(h) \in \mathcal{D}$ and $[\texttt{object}] \sqsubseteq_{len} evalStackT_{filterStart(h)}$. In particular, this means that (**pc**) holds in the new state and that $evalStackT_{filterStart(h)} = [\texttt{object}]$. From (**wt4**) applied to the *Throw* instruction that threw *exc*, we know that the type of *exc* is a subtype of `object`. Since $evalStack = [exc]$, it follows that the invariants (**stack1**) and (**stack2**) hold in the new state.

In all the other cases, EXCCLR executes submachines that do not update *pc* or *evalStack*, so that the claim follows from the induction hypothesis.

Subcase 2.2.2 $existsHanWithinFrame(s)$ is false. Then SEARCHINVFRAME is executed, without affecting any of the invariants. □

Remark From the proof point of view, the case when a method raises an exception is treated as if the corresponding call instruction in the invoker frame would have thrown the exception. Similarly, the case when a class initialization (that might happen at any time if the class is `beforefieldinit`) throws a `TypeInitializationException` is considered as if the instruction executed just before the initialization would have thrown the `TypeInitializationException`. Both cases could also be treated (modulo the corresponding exception object on the *evalStack*) as if the current instruction would be *Throw*.

11 CONCLUSION

We have defined an abstract model for the CLR exception handling mechanism. It lays the ground for a mathematical correctness proof of the CLR bytecode verifier. Through a mathematical analysis we discovered a few gaps in the ECMA standard for CLR. Our model fills these gaps.

12 ACKNOWLEDGMENT

We are thankful to Jonathan Keljo for the useful discussion and to the three referees for valuable questions.

References

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

[2] The ThreadAbortException class (System.Threading). Web pages at http://msdn2.microsoft.com/library/system.threading.threadabortexception.aspx.

[3] Egon Börger. The ASM Ground Model Method as a Foundation for Requirements Engineering. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume Springer, Lecture Notes in Computer Science 2772, pages 145–160, 2003.

[4] Egon Börger. The ASM Refinement Method. *Formal Asp. Comput.*, 15(2-3):237–257, 2003.

[5] Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. A High–Level Modular Definition of the Semantics of C♯. *Theoretical Computer Science*, 336(2–3):235–284, June 2005.

[6] Egon Börger and Wolfram Schulte. A Practical Method for Specification and Analysis of Exception Handling-A Java/JVM Case Study. *IEEE Trans. Softw. Eng.*, 26(9):872–887, 2000.

[7] Egon Börger and Robert F. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer–Verlag, 2003.

[8] Egon Börger and Robert F. Stärk. Exploiting Abstraction for Specification Reuse: The Java/C♯ Case Study. In F. S. de Boer et al., editors, *Formal Methods for Components and Objects: Second International Symposium, FMCO 2003, Leiden, The Netherlands*, pages 42–76. Springer–Verlag, Lecture Notes in Computer Science 3188, 2004.

[9] Chris Brumme. The Exception Model. Web pages at http://blogs.msdn.com/cbrumme/, 2003.

[10] Common Language Infrastructure (CLI) – Standard ECMA–335, 2005. Third Edition.

[11] Nicu G. Fruja. The Soundness of the .NET CLR Bytecode Verifier. submitted.

[12] Nicu G. Fruja. Experiments with CLR. available at http://www.inf.ethz.ch/personal/fruja/publications/clrexctests.pdf, November 2004. Example programs to determine the meaning of CLR features not specified by the ECMA standard.

[13] Nicu G. Fruja. Specification and Implementation Problems for C♯. In W. Zimmermann and B. Thalheim, editors, *11th International Workshop on Abstract State Machines, ASM 2004, Wittenberg, Germany*, pages 127–143. Springer–Verlag, Lecture Notes in Computer Science 3052, 2004.

[14] Nicu G. Fruja. The Correctness of the Definite Assignment Analysis in C♯. In Vaclav Skala and Piotr Nienaltowski, editors, *2nd International .NET Technologies Workshop, Plzen, Czech Republic*, pages 81–88, 2004.

[15] Nicu G. Fruja. The Correctness of the Definite Assignment Analysis in C♯ (**Extended Version**). *J. Object Technology*, 3(9):29–52, 2004.

[16] Nicu G. Fruja. A Modular Design for the .NET CLR Architecture. In A. Slissenko D. Beauquier and E. Börger, editors, *12th International Workshop on Abstract State Machines, ASM 2005, Paris, France*, pages 175–199, March 2005.

[17] Nicu G. Fruja. *Type Safety in* C♯ *and .NET CLR*. PhD thesis, ETH Zürich, expected 2007. in preparation.

[18] Nicu G. Fruja and Egon Börger. Analysis of the .NET CLR Exception Handling. In Vaclav Skala and Piotr Nienaltowski, editors, *3rd International Conference on .NET Technologies, .NET 2005, Pilsen, Czech Republic*, pages 65–75, June 2005.

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

[20] Horatiu V. Jula and Nicu G. Fruja. An Executable Specification of C♯. In A. Slissenko D. Beauquier and E. Börger, editors, *12th International Workshop on Abstract State Machines, ASM 2005, Paris, France*, pages 275–287. University Paris 12, March 2005.

[21] Christian Marrocco. An Executable Specification of the .NET CLR, March 2005. Diploma Thesis supervised by N. G. Fruja.

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

[23] Robert F. Stärk. Formal Specification and Verification of the C♯ Thread Model. *Theor. Comput. Sci.*, 343(3):482–508, 2005.

[24] Robert F. Stärk and Egon Börger. An ASM Specification of C♯ Threads and the .NET memory model. In W. Zimmermann and B. Thalheim, editors, *11th International*

Workshop on Abstract State Machines, ASM 2004, Wittenberg, Germany, pages 38–60. Springer–Verlag, Lecture Notes in Computer Science 3052, 2004.

[25] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine – Definition, Verification, Validation*. Springer–Verlag, 2001.

ABOUT THE AUTHORS

Nicu G. Fruja is a PhD student and research assistant at the Computer Science Department at ETH Zürich, Switzerland. His research interests are in the areas of formal methods, specification, verification and validation of systems, abstract state machines, semantics of programming languages, bytecode verification. He can be reached at fruja@inf.ethz.ch. See also http://www.inf.ethz.ch/personal/fruja/.

Egon Börger Professor of Computer Science at University of Pisa, Italy. (Co-) Author of four books and of over 100 research papers in logic and computer science. Current research interest in rigorous methods and their industrial applications for the design and the analysis of hardware/software systems.