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
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:
2 PRELIMINARIES

\[ f(s) := t \] update \( f \) at \( s \) to \( t \)
\[ \text{if } \varphi \text{ then } P \text{ else } Q \] if \( \varphi \), then execute \( P \), else \( Q \)
\[ P \quad Q \] execute \( P \) and \( Q \) in parallel
\[ \text{let } x = t \text{ in } P \] assign \( t \) to \( x \) and then execute \( P \)
\[ P \quad \text{seq} \quad Q \] execute \( P \) and then \( Q \)
\[ P \quad \text{or} \quad Q \] 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., \texttt{push}, \texttt{pop}, \texttt{top}, \texttt{length}, and \texttt{·})\(^1\), we use \texttt{split}(\(L\), \(1\)) to split off the last element of the list \(L\), i.e., \texttt{split}(\(L\), \(1\)) is the pair \((L'\cdot[x])\) 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 \textit{StackWalk} pass of the exception mechanism. The other two passes, \textit{Unwind} and \textit{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 \texttt{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 \(locAdr : \text{Map}(\text{Local}, \text{Adr})\), arguments addresses \(argAdr : \text{Map}(\text{Arg}, \text{Adr})\), an evaluation stack\(^2\) \texttt{evalStack} : \text{List}(\text{Value})\), and a method reference \(meth : \text{MRef}\). The \textit{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 \textit{frameStack} and is defined as a list of frames. Note that we separate the current frame from the stack of call frames, i.e., \textit{frame} is not contained in \textit{frameStack}. The macros \texttt{PUSHFRAME} and \texttt{POPFRA}ME are used to push and pop the \textit{frame}, respectively.

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

\(^1\)The “·” denotes the operation \texttt{append} for lists.
\(^2\)In order to simplify the exposition we describe here the \textit{evalStack} as a list of values though [16] defines it as a list of pairs from \texttt{Value} \times \texttt{Type}.
Fig. 1 The CLR\(_E\) machine

\[ \text{CLR}_E \equiv \begin{cases} \text{if } \text{switch} = \text{ExcMech} \text{ then} & \text{EXECCLR} \\ \text{elseif } \text{switch} = \text{Noswitch} \text{ then} & \text{INITIALIZECLASS or EXECCLR}_E(\text{code}(pc)) \end{cases} \]

\text{POPFRAME} \equiv \begin{cases} \text{let } (\text{frameStack}', [(pc', \text{locAdr}', \text{argAdr}', \text{evalStack}', \text{meth}'])] = \text{split}(	ext{frameStack}, 1) \text{ in} \\ \text{pc} := pc' \\ \text{locAdr} := \text{locAdr}' \\ \text{argAdr} := \text{argAdr}' \\ \text{evalStack} := \text{evalStack}' \\ \text{meth} := \text{meth}' \\ \text{frameStack} := \text{frameStack}' \end{cases} \]

3 THE GLOBAL MACHINE STRUCTURE

Every time an exception occurs, the control is transferred from “normal” execution (in 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 EXECCLR\(_E\) and triggers the execution of EXCCLR. The machine EXECCLR\(_E\) is an extension of the exception-handling-free machine 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 occurred exception is called an *inner exception*. As we will see below, the outer exception is not discarded but its context is saved by **EXC** CLR 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 *exHA* 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** 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 of 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 **EXC** CLR 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.

---

4Currently, 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

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>isInTry(pos, h)</td>
<td>tryStart(h) ≤ pos &lt; tryStart(h) + tryLength(h)</td>
</tr>
<tr>
<td>isInHandler(pos, h)</td>
<td>handlerStart(h) ≤ pos &lt; handlerStart(h) + handlerLength(h)</td>
</tr>
<tr>
<td>isInFilter(pos, h)</td>
<td>filterStart(h) ≤ pos &lt; handlerStart(h)</td>
</tr>
</tbody>
</table>

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

\[
\text{ClauseKind} = \text{catch} \mid \text{filter} \mid \text{finally} \mid \text{fault}
\]

\[
\text{Exc} = \text{Exc} (\text{clauseKind} : \text{ClauseKind}, \text{tryStart} : \text{Pc}, \text{tryLength} : \text{N}, \text{handlerStart} : \text{Pc}, \text{handlerLength} : \text{N}, \text{type} : \text{ObjClass}, \text{filterStart} : \text{Pc})
\]

Any 7-tuple of the above form describes a handler of kind clauseKind which “protects” the region 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 filterStart < handlerStart. We often refer to the sequence of instructions between filterStart and 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 pos ∈ Pc and a handler h ∈ Exc. Note that if the predicate isInFilter is true, then filterStart is defined and therefore h is of kind filter. Based on

\footnote{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.

| `ExcRec`       | `ExcRec` ( | `exc` : `ObjRef` |
|               | `pass` : `{StackWalk,Unwind}` |
|               | `stackCursor` : `Frame × N` |
|               | `handler` : `Frame × N` |

| `LeaveRec`     | `LeaveRec` ( | `pass` : `{Leave}` |
|               | `stackCursor` : `Frame × N` |
|               | `target` : `Pc` |

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.

\[
passRecStack : \text{List}(\text{ExcRec} \cup \text{LeaveRec}) \quad \text{passRecStack} = []
\]

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:

\[
\.try \{ \\
\quad \text{block} \\
\} \quad \text{catch block}_1 \\
\} \quad \text{catch block}_2 \quad \Rightarrow \quad \text{try} \{ \\
\} \quad \text{try} \{ \\
\quad \text{block} \\
\} \quad \text{catch block}_1 \\
\} \quad \text{catch block}_2
\]

We can now summarize the overall behavior of excCLR, which is defined in Fig. 3 and analyzed in detail in the following sections, by saying that if there is a handler in the frame defined by stackCursor, then excCLR will try to find (when StackWalking) or to execute (when Unwinding) or to leave (when Leafing) 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 \( \text{excHA} \) of handlers of the corresponding method \( m \).

\[
\text{existsHanWithinFrame}((\_\_\_\_\_\_m), n) \iff n < \text{length}(\text{excHA}(m))
\]

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

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

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

- a \texttt{catch} handler whose \texttt{try} block protects the program counter \( pc \) of the frame pointed at by \( \text{stackCursor} \) and whose \texttt{type} is a supertype of the exception type;

\[
\text{matchCatch}(pos,t,h) \iff \text{isInTry}(pos,h) \land \text{clauseKind}(h) = \text{catch} \land t \leq \text{type}(h)
\]

- a \texttt{filter} handler whose \texttt{try} block protects the \( pc \) of the frame pointed at by \( \text{stackCursor} \);

\[
\text{matchFilter}(pos,h) \iff \text{isInTry}(pos,h) \land \text{clauseKind}(h) = \text{filter}
\]

- a \texttt{filter} handler whose \texttt{filter} region contains the \( pc \) of the frame pointed at by \( \text{stackCursor} \). This corresponds to an outer exception described below.

The order of the \texttt{if} clauses in the \texttt{let} statement from the rule \texttt{StackWalk} in Fig. 3 is not important. This is justified by the following property:

**Disjointness 2** For every type \( t \), the predicates \( \text{matchCatch}^t \), \( \text{matchFilter} \) and \( \text{isInFilter} \) are pairwise disjoint.

\(^6\)By \( \text{matchCatch}^t \) we understand the predicate defined by the set \( \{(pos,h) \mid \text{matchCatch}(pos,t,h)\} \).
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:

\[
\text{GoToNxtHan} \equiv \text{stackCursor} := (fr, n + 1) \\
\text{where stackCursor} = (fr, n)
\]

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 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.

\[
\text{FoundHandler} \equiv \\
\text{pass} := \text{Unwind} \\
\text{handler} := \text{stackCursor} \\
\text{Reset}(s, fr) \equiv s := (fr, 0)
\]

**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 EXECCLR_E.

---

We use the actualTypeOf function defined in [16] to determine the runtime type of the exception.
Fig. 3 The exception handling machine EXCCLR

\[ \text{EXCCLR} \equiv \text{match pass} \]

**StackWalk** → 
if existsHanWithinFrame(stackCursor) then
  let \( h = \text{hanWithinFrame}(\text{stackCursor}) \) 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
    GOTOONXTHAN
  else
    SEARCHINVFRAME(fr)
    where stackCursor = (fr, -) and fr = (pos, -,-,-,-)

**Unwind** → 
if existsHanWithinFrame(stackCursor) then
  let \( h = \text{hanWithinFrame}(\text{stackCursor}) \) in
  if matchTargetHan(handler, stackCursor) then
    EXEC Han(h)
  elseif matchFinFault(pc, h) then
    EXEC Han(h)
    GOTOONXTHAN
  elseif isInHandler(pc, h) then
    ABORT PREV PASS REC
    GOTOONXTHAN
  elseif isInFilter(pc, h) then
    CONTINUE OUTER EXC
  else
    GOTOONXTHAN
  else
    POPFRAME
    SEARCHINVFRAME(frame)

**Leave** → 
if existsHanWithinFrame(stackCursor) then
  let \( h = \text{hanWithinFrame}(\text{stackCursor}) \) in
  if isFinFromTo(h, pc, target) then
    EXEC Han(h)
  elseif isRealHanFromTo(h, pc, target) then
    ABORT PREV PASS REC
    GOTOONXTHAN
  else
    pc := target
    evalStack := []
    POPREC
    switch := Noswitch

\[ \text{EXECFILTER}(h) \equiv pc := \text{filterStart}(h) \]
\[ \text{evalStack} := [\text{exc}] \]
\[ \text{locAdr} := \text{locAdr}' \]
\[ \text{argAdr} := \text{argAdr}' \]
\[ \text{meth} := \text{meth}' \]
\[ \text{PUSHFRAME} \]
\[ \text{switch} := \text{Noswitch} \]
\[ \text{where stackCursor} = ((-, \text{locAdr}', \text{argAdr}', -, \text{meth}'), -) \]
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 occurred in yet another *filter* region of a nested handler (see the *isInFilter* clause), then the given *filter* region continues executing normally (after the exception has been taken care of);

- if the inner exception is not taken care of in the *filter* region, then it will be discarded (via the *CONTINUEOUTEREXEC* 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 *EXITINNEREXEC* 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 occurred is reached.

```
EXITINNEREXEC ≡
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 *Unwinding*, 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″)) ⇔ sameFrame(fr′, fr″) ∧ n′ = n″
sameFrame(fr′, fr″) ⇔ pr_i(fr′) = pr_i(fr″), ∀i ∈ {2, 3, 5}
```
• a matching `finally/fault` handler whose associated `try` block protects the `pc`;

\[
\text{matchFinFault}(pos, h) \iff \text{isInTry}(pos, h) \land \text{clauseKind}(h) \in \{\text{finally, fault}\}
\]

• 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 `EXECCHAN`: the `pc` is set to the beginning of the handler region, the exception reference is loaded on the evaluation stack (when `EXECCHAN` is applied for executing `finally/fault` handler regions, nothing is pushed onto `evalStack`), and the control switches to `EXECCLR`.

\[
\text{EXECCHAN}(h) \equiv \\
\begin{align*}
\text{pc} & : = \text{handlerStart}(h) \\
\text{evalStack} & : = \begin{cases} 
\text{if } \text{clauseKind}(h) \in \{\text{catch, filter}\} & \text{then } [\text{exc}] \\
\text{else } [] \end{cases} \\
\text{switch} & : = \text{Noswitch}
\end{align*}
\]

**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 occurred in a `filter` region of a nested handler), then the handler region continues executing normally (after the exception is taken care of);

• if the exception is not taken care of in the handler region, i.e, escapes the handler region, then the following two actions are taken:
  – the previous pass of `excCLR` is aborted through `ABORTPREVPASSREC`;

  ```
  ABORTPREVPASSREC \equiv \text{pop}(\text{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 occurred 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 sequentially 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 CONTINUEOUTEREXEC is executed.

\[
\text{CONTINUEOUTEREXEC} \equiv \\
\text{PopFrame} \\
\text{PopRec seq GoToNxtHan}
\]

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

\[
\text{SetRecUndef} \equiv \text{exc} := \text{undef} \\
\text{pass} := \text{undef} \\
\text{stackCursor} := \text{undef} \\
\text{target} := \text{undef} \\
\text{handler} := \text{undef}
\]

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\(^\text{\#}\) 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.

\[
\text{isFinFromTo}(h, pos', pos'') \iff \text{clauseKind}(h) = \text{finally} \land \text{isInTry}(pos', h) \land \neg\text{isInTry}(pos'', h) \land \neg\text{isInHandler}(pos'', h)
\]

\[
\text{isRealHanFromTo}(h, pos', pos'') \iff \text{clauseKind}(h) \in \{\text{catch, filter}\} \land \text{isInHandler}(pos', h) \land \neg\text{isInHandler}(pos'', h)
\]

\(^8\)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 EXCHAN 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 EXCCLR_E execution (see Fig. 3).

8 THE RULES OF EXCCLR_E

The rules of EXCCLR_E in Fig. 4 specify the effect of the CIL instructions related to exceptions. Each of these rules transfers the control to EXCCLR. Throw pops the topmost evaluation stack element (see Remark below), which is supposed to be an exception reference. It loads 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 ≠ undef, then the current pass record is pushed onto passRecStack.

\[
\text{LOADREC}(r) \equiv \\
\begin{align*}
\text{if } r \in \text{ExcPass} \text{ then} \\
\quad \text{let } (\text{exc}', \text{pass}', \text{stackCursor}', \text{handler}') = r \text{ in} \\
\quad \text{exc} & \quad := \text{exc}' \\
\quad \text{pass} & \quad := \text{pass}' \\
\quad \text{stackCursor} & \quad := \text{stackCursor}' \\
\quad \text{handler} & \quad := \text{handler}' \\
\quad \text{else let } (\text{pass}', \text{stackCursor}', \text{target}') = r \text{ in} \\
\quad \text{pass} & \quad := \text{pass}' \\
\quad \text{stackCursor} & \quad := \text{stackCursor}' \\
\quad \text{target} & \quad := \text{target}' \\
\quad \text{if } \text{pass} \neq \text{undef} \text{ then PUSHREC}
\end{align*}
\]

\[
PUSHREC \equiv \\
\begin{align*}
\text{if } \text{pass} = \text{Leave} \text{ then push}(\text{passRecStack}, (\text{pass}, \text{stackCursor}, \text{target})) \\
\quad \text{else push}(\text{passRecStack}, (\text{exc}, \text{pass}, \text{stackCursor}, \text{handler}))
\end{align*}
\]

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
defined by the following code template\(^9\):

\[
\text{RAISE}(c) \equiv \text{NewObj}(c :: \text{ctor})
\]

\[
\text{Throw}
\]

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 \(c :: \text{ctor}\) 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 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

---

10One 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 $\text{ExcRec}$ of exception records is refined to include an object reference denoted $\text{discardedTAE}$.

$$
\text{ExcRec} = \text{ExcRec} ( \text{exc} : \text{ObjRef} \\
\text{pass} : \{\text{StackWalk,Unwind}\} \\
\text{stackCursor} : \text{Frame} \times \mathbb{N} \\
\text{handler} : \text{Frame} \times \mathbb{N} \\
\text{discardedTAE} : \text{ObjRef} )
$$

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

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

- The macro $\text{ABORTPREVPASSREC}$ used in the $\text{isInHandler}$ clause of the $\text{Unwind}$ rule is refined to also “transfer” the abort request (if any), i.e., the current exception will take over the $\text{ThreadAbortException}$ reference (if any) carried by the discarded exception record:

$$
\text{ABORTPREVPASSREC} \equiv \\
\text{pop}(\text{passRecStack}) \\
\text{let } r = \text{top}(\text{passRecStack}) \text{ in} \\
\text{if } r \in \text{ExcRec} \text{ then} \\
\text{let } (\_,\_,\_,\_,\_, \text{discardedTAE'}) = r \text{ in} \\
\text{if } \text{discardedTAE'} \neq \text{undefined} \text{ then} \\
\text{discardedTAE} := \text{discardedTAE'}
$$

Similarly, $\text{POPREC}$ used in the $\text{isInFilter}$ clause of the $\text{Unwind}$ rule is refined to “transfer” the $\text{ThreadAbortException}$ that has to be raised later again. Note that in this way a $\text{ThreadAbortException}$ can escape a filter region.

Also, the macro $\text{STRECUndef}$ is refined to also reset $\text{discardedTAE}$ to $\text{undefined}$. 
• The isRealHanFromTo clause of the Leave rule in Fig. 3 is modified to rethrow the discarded ThreadAbortException:

```plaintext
let r = top(passRecStack) in
let (_, _, _, discardedTAE') = r in
if discardedTAE' ≠ undefined then
    POPREC seq
    exc := discardedTAE'
    pass := StackWalk
    stackCursor := (frame, 0)
    handler := undefined
    discardedTAE := discardedTAE'
else pop(passRecStack)
```

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 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 (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

\[
\text{check}(\text{meth}, \text{pos}, \text{evalStackT}) \iff \text{match code(pos)}
\]

\begin{align*}
\text{Throw} & \rightarrow \text{top(\text{evalStackT})} \subseteq \text{object} \\
\text{Rethrow} & \rightarrow \text{True} \\
\text{EndFilter} & \rightarrow \text{evalStackT} = [\text{int32}] \\
\text{EndFinally} & \rightarrow \text{True} \\
\text{Leave} & \rightarrow \text{True}
\end{align*}

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 with every reachable instruction. The type stack state \text{evalStackT} specifies for each slot of \text{evalStack} a required type in that slot and thereby also the number of values on the \text{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 \text{check} the conditions checked by the verifier for the instructions related to exceptions. The checks for a single instruction operate on \text{evalStackT}. The relation \subseteq denotes the compatibility relation between types; for a formal definition see [11].

In JVM, the \text{Throw} instruction expects an object of type \text{Throwable} on the stack. The CLR bytecode verifier is not so strict: it requires that the top stack element is of type \text{object}. The \text{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 \text{Rethrow}, \text{EndFinally}, and \text{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 \text{pos} yields constraints on how to match the type stack states of all instructions that are runtime possible control-flow successors of \text{pos}. In Fig. 6, we define the function \text{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 \text{ExecutionEngineException} may be thrown at any time during the execution of a program. Therefore, for an instruction at \text{pos}, all the handlers \( h \) that protect \text{pos}

\footnote{The ECMA standard uses the name \text{stack state}. However, we prefer type stack state since we have a more complex notion of state and also more than one stack.}
The type stack state successors of the instructions related to exceptions are included into the set of possible successor type stack states by means of a function \( \text{excHandlers}(\text{meth}, \text{pos}) \). Upon entering a catch handler, the type stack state contains only the type \( \text{type}(h) \) 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 [\text{object}]. Except for the case of a filter region, the successor code index is given by \( \text{handlerStart}(h) \). In case of a filter region, the successor is \( \text{filterStart}(h) \).

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

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

Thus, target is an element of \( \text{LeaveThroughFin}(\text{meth}, \text{pos}) \) for an \( \text{EndFinally} \) instruction at \( \text{pos} \) if this \( \text{EndFinally} \) 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 $\sqsubseteq_{\text{len}}$ of pointwise compatibility of type stack states is defined for lists of types $L', L''$ of lengths $m, n$ as follows:

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

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

1. The elements of $\mathcal{D}$ are valid code indices of mref.
2. $0 \in \mathcal{D}$.
3. $\text{evalStack}_T_0 = [\,]$.
4. If $i \in \mathcal{D}$, then check(mref, i, evalStack$_T_i$) is true.
5. If $i \in \mathcal{D}$ and $(j, \text{evalStack}'_T)$ $\in$ succ(mref, i, evalStack$_T_i$), then $j \in \mathcal{D}$ and $\text{evalStack}'_T \sqsubseteq_{\text{len}} \text{evalStack}_T_j$.

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

$(wt1)$ The elements of $\mathcal{D}$ are valid code indices of mref.
$(wt2)$ $0 \in \mathcal{D}$.
$(wt3)$ $\text{evalStack}_T_0 = [\,]$.
$(wt4)$ If $i \in \mathcal{D}$, then check(mref, i, evalStack$_T_i$) is true.
$(wt5)$ If $i \in \mathcal{D}$ and $(j, \text{evalStack}'_T)$ $\in$ succ(mref, i, evalStack$_T_i$), then $j \in \mathcal{D}$ and $\text{evalStack}'_T \sqsubseteq_{\text{len}} \text{evalStack}_T_j$.  

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 EXECCLR_N to CLR_E.
The soundness theorem proved for the exception-free machine EXECCLR_N guarantees
that the following type-safety invariants hold at runtime for well-typed methods.

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

(stack1) the current \( evalStack \) has the same length as \( evalStack_{pc} \);

(stack2) the values on the current \( evalStack \) are compatible with the corresponding types
assigned in \( evalStack_{pc} \);

(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 Theorem 1, the invariants (loc)-(box) are not
affected by computation steps of EXECCLR. 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) \( pc \in D \)

(stack1) \( length(evalStack) = length(evalStack_{pc}) \)

(stack2) \( \vdash evalStack(j) : evalStack_{pc}(j) \), for every \( j < length(evalStack) \)
Proof. The proof assumes the proof for \( \text{EXECCLR}_N \) and proceeds by induction on the run of \( \text{CLR}_E \). In the initial state of \( \text{CLR}_E \), the invariants for the single existing frame (of the \texttt{.entr}ypoint method) are satisfied. In fact, this frame satisfies \( pc = 0 \) and \( \text{evalStack} = [] \). Thus, \((pc)\) holds by \((\text{wt2})\) and \((\text{stack1})\), and \((\text{stack2})\) follows from \((\text{wt3})\). In the induction step, we proceed by case distinction on \( code(pc) \), whether \( \text{EXECCLR}_E \) has the control \((\text{switch} = \text{Noswitch})\) or \( \text{EXECCLR} \ (\text{switch} = \text{ExcMech}). \) 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 \( \text{switch} = \text{Noswitch} \). The subcases \( code(pc) \in \{ \text{Throw, Rethrow, EndFinally} \} \) are trivial because they do not affect neither \( pc \) nor \( \text{evalStack} \).

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

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

Case 2 \( \text{switch} = \text{ExcMech} \). We present two subcases out of three.

Subcase 2.1 \( pass = \text{ExcMech} \). Let \( s \) and \( \text{target} \) be the current value of the \( \text{stackCursor} \) and the target associated to this \( \text{Leave} pass \), respectively.

Subcase 2.1.1 \( \text{existsHanWithinFrame}(s) \) is true. Let \( h \) be the handler pointed to by \( s \).

If \( \text{isRealHanFromTo}(h, pc, target) \) is true, \( \text{EXECCLR} \) executes \text{ABORTPREVPASSREC} and \text{GOTONXTHA} which do not influence the invariants. Therefore, the claim follows from the induction hypothesis.

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

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

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

If \( code(pc) \) is an instruction \( \text{Leave(target)} \), then by the definition of Subcase 2.1.2, the set \( \{ h \in \text{excHA}(\text{meth}) \mid \text{isFinFromTo}(h, pc, target) \} \) is empty. Then the definition of \( \text{succ} \) implies \((\text{target, []}) \in \text{succ}(\text{meth, pc, evalStack}_p) \).
Thus, in every case holds \((\text{target, []}) \in \text{succ}(\text{meth, } \text{pc, evalStackT}_{\text{pc}})\). This together with \((\text{pc})\) and \((\text{wt5})\) implies \(\text{target} \in \mathcal{D}\) and \([\cdot] \sqsubseteq \text{len evalStackT}_{\text{target}}\). Consequently, the invariants \((\text{pc})\), \((\text{stack1})\), and \((\text{stack2})\) hold for the current frame (the last two hold since \(\text{evalStackT}_{\text{target}}\) shall be \([\cdot]\)).

**Subcase 2.2** \(\text{pass} = \text{StackWalk}\). Let \(s\) be the current value of the \text{stackCursor}.

**Subcase 2.2.1** \(\exists \text{HanWithinFrame}(s)\) is true. Let \(h\) be the handler pointed to by \(s\) and \(\text{pos}\) the program counter of the frame pointed to by \(s\).

If \(h\) satisfies \(\text{matchFilter}(\text{pos}, h)\), then \(\text{EXCCLR}\) executes \(\text{EXECFILTER}(h)\) which loads on the \text{frameStack} a frame with \(\text{pc}\) set to \(\text{filterStart}(h)\) and \(\text{evalStack}\) to \([\text{exc}]\). From the definition of \(\text{matchFilter}\), we have \(\text{isInTry}(\text{pos}, h)\) and \(\text{clauseKind}(h) = \text{filter}\). By this and the definitions of \(\text{succ}\) and \(\text{exchandlers}\), we get \((\text{filterStart}(h), \text{object}) \in \text{succ}(\text{mref}, \text{pos}, \text{evalStackT}_{\text{pos}})\) where \(\text{mref}\) is the method of the frame pointed to by \(s\). The definition of \(\text{EXECFILTER}\) implies that \(\text{mref}\) is also the method of the new current frame. Thus, \((\text{filterStart}(h), \text{object}) \in \text{succ}(\text{meth}, \text{pos}, \text{evalStackT}_{\text{pos}})\). This, \((\text{pc})\), and \((\text{wt5})\) imply \(\text{filterStart}(h) \in \mathcal{D}\) and \(\text{object} \sqsubseteq \text{len evalStackT}_{\text{filterStart}(h)}\). In particular, this means that \((\text{pc})\) holds in the new state and that \(\text{evalStackT}_{\text{filterStart}(h)} = \text{object}\).

From \((\text{wt4})\) applied to the \text{Throw} instruction that threw \text{exc}, we know that the type of \text{exc} is a subtype of \text{object}. Since \(\text{evalStack} = [\text{exc}]\), it follows that the invariants \((\text{stack1})\) and \((\text{stack2})\) hold in the new state.

In all the other cases, \(\text{EXCCLR}\) executes submachines that do not update \(\text{pc}\) or \(\text{evalStack}\), so that the claim follows from the induction hypothesis.

**Subcase 2.2.2** \(\exists \text{HanWithinFrame}(s)\) is false. Then \(\text{SEARCHINVFRAME}\) is executed, without affecting any of the invariants. \(\square\)

**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 \text{beforefieldinit}) throws a \text{TypeInitializationException} is considered as if the instruction executed just before the initialization would have thrown the \text{TypeInitializationException}. Both cases could also be treated (modulo the corresponding exception object on the \text{evalStack}) as if the current instruction would be \text{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


12 ACKNOWLEDGMENT


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.