






Modular Transformation of Java Exceptions Modulo Errors

Robert Rubbens^(✉) , Sophie Lathouwers , and Marieke Huisman 

Formal Methods and Tools, University of Twente, Enschede, The Netherlands
r.b.rubbens@utwente.nl

Abstract. Deductive verifiers are used more and more in both academia and industry to prevent costly bugs. Their capabilities of verifying concurrent programs are getting better, but they are still lagging behind with regard to many major programming language features such as exceptions. To improve the situation, this work presents a semantics of Java exceptions which reduces the annotation burden on the user, while still allowing verification of exceptions. This is accomplished by ignoring sources of errors which are irrelevant to functional verification. Additionally, to deal with the complex control flow introduced by `finally`, a transformation is proposed that simplifies verification of exceptional postconditions and `finally` into postconditions and `goto`. We implement the approach and evaluate it against several common exception patterns.

Keywords: Deductive verification · Java · VerCors · Exceptions · Finally · Errors

1 Introduction

For programs which require high reliability and robustness, such as nuclear power plant, railroad, or tunnel software, bugs are not acceptable. To ensure that a program complies with the highest standards of correctness, deductive verifiers have been developed. Deductive verifiers implement logics to reason about programs mathematically, and can ensure adherence to a specification. This guarantee increases the chance that bugs will be caught before software is deployed.

If we have tools that can verify if a program is free of bugs, why do we still have bugs? Part of the answer is that industry uses language features that are often unsupported by deductive verifiers. An example of such a feature is the Java exceptions mechanism, which is the primary tool to identify and handle failures of many kinds in Java code. Osman et al. indicate that for four mature Java projects the proportion of exception-related code remains around 1%, even after 6 years of ongoing development [27]. For code bases like Hadoop and Tomcat, which contain millions of lines of code, these are significant numbers [4, 5]. We do not know of any efforts to fully verify code bases such as these, but to accomplish this, support for exceptions is mandatory.

There are several projects that allow verification of Java, and some support exceptions. For example, OpenJML [7] can verify sequential Java. Another example, VerCors [2], can verify concurrent Java, but does not have support for exceptions at all. Finally, Verifast [19] can verify concurrent Java with exceptions, but does not support `finally`. Therefore, when verifying Java, a choice must be made. Either sequential Java can be verified with full support for exceptions, or concurrent Java can be verified with limited exception support. What is surprising is that this dichotomy is not necessary: concurrent execution and exceptional control flow are orthogonal concerns.

In this work, we try to improve the state of the art by implementing full support for exceptions in VerCors, a verifier of concurrent Java.

Java itself has some facilities for checking at compile time if some exceptions are handled. Particularly, “checked exceptions” are required to be handled when they occur. “Unchecked exceptions” are not required to be handled. Exceptions are intended to make error handling more structured and robust, but there are signs they currently fail at the latter. According to a study done by Sena et al. 20% of the bugs in 656 Java projects are related to improper exception usage [31].

One way of solving this is only using checked exceptions, as Java requires each checked exception to be handled. Unfortunately, Java ignores unchecked exceptions, so this rule is easily broken. Furthermore, various parts of the standard library use unchecked exceptions, so it is easy to break this rule accidentally, and hard to manually ensure only checked exceptions are used. This is where deductive verifiers can help: verifying exception handling code automatically could help with reducing bugs related to exception handling, as the verifier can guarantee that an exception is always handled correctly.

Verifying exceptions poses three problems. First, supporting `finally` entails handling complex control flow. To avoid a monolithic implementation, a modular transformation must be designed that decomposes the control flow as much as possible. Second, according to *The Java Language Specification* [14] (JLS), exceptions can come from many places, not just the `throw` statement, but also from e.g. allocating memory. Requiring the user to create annotations for all these cases is unfeasible. A subset of Java exceptions must be chosen such that the annotation burden is reduced, while still allowing verification of common exception patterns. Third, standard library code that throws checked and unchecked exceptions must be annotated with exceptional specifications.

In this work, we try to resolve the first and second problem, and leave the third for future work. First, to decompose complex control flow introduced by exceptions, we transform all control flow in the program into exceptions. The exceptional control flow is then transformed into `goto` statements. This approach splits up the transformation into multiple steps, making it more modular. It also reduces the number of different kinds of control flow, which simplifies the semantics in the intermediate stages.

Second, to relieve the user of the annotation burden, we define a subset of Java exceptions called “exceptions modulo errors”. This view allows exceptions to originate from `throw` statements and method calls with `throws` attributes,

and ignores exceptions caused by memory allocation failures or other low-level implementation details. Reducing the annotation burden this way has a cost: guarantees of verification are weaker because some errors are ignored. However, since the assumption of exceptions modulo errors is often made in commercial software development, we argue it is a reasonable simplification.

Contributions. The main contributions of this work are:

- A simplified semantics of exceptions allowing verification of functional properties which ignores a number of specific errors.
- An evaluation of the simplified semantics with common exception patterns.
- An encoding of exceptional postconditions and `finally` into postconditions and `goto`.
- An implementation of support for exceptions in the VerCors verifier.

The files used for evaluating exceptions modulo errors, as well as instructions for running jStar, Krakatoa, and VerCors, can be found in the package accompanying this paper. The package can be found here: [30].

Paper Structure. Section 2 discusses the background on Java verification in VerCors. Section 3 discusses related work. Section 4 discusses the definition of exceptions modulo errors. Section 5 discusses how `finally` complicates Java verification, and how this can be resolved by transforming all control flow into exceptional control flow. Section 6 evaluates the approach presented in this work against common exception patterns. Section 7 reflects on the presented approach. Section 8 contains the conclusions and future work.

2 Background

This section discusses background necessary to understand how VerCors verifies Java programs. We first discuss the notion of abrupt termination in Java. Then we discuss how VerCors verifies Java programs.

2.1 Abrupt Termination

Abrupt termination [22, p. 14] is a grouping term for control flow that does not go from one statement to the next, like regular control flow. Instead, abrupt termination is when a statement terminates not because it is completed, but because it is terminated sooner than normal and control flow is redirected to another program point. Abrupt termination is sometimes also referred to as non-local or non-linear control flow.

One example of abrupt termination is the `throw` statement, as it aborts execution of the current block and redirects control flow to the nearest `catch` block. Other abrupt termination keywords are: `break`, `continue` and `return`. They all terminate the current block earlier than normal, and redirect control flow to another program point.

The labelled `break` and `continue` statements are an extra source of complexity as they allow the user to specify which loop to `break` from or `continue`. These constructs can be useful when nested loops are used. Furthermore, labelled `break` can also be used within `if`, `switch` or labelled blocks.

2.2 VerCors

VerCors is verifier for concurrent software [2]. It can verify programs written in Java, OpenCL, C, and PVL. VerCors uses separation logic to reason about concurrent access to data.

It is a deductive verifier, which means it uses a system of proof rules to establish correctness of the input. When VerCors reports an input program to be correct, it means it has found a proof using logical inference. For more information about deductive verification, we refer the reader to “Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools” by Hähnle and Huisman [16].

It is also a modular verifier, which means that verification of each method only depends on the contract of other methods, and not on their implementation. This also holds for concurrency: threads are verified “thread-modularly”, which implies that adding another thread does not invalidate the correctness of previously verified threads.

Figure 1 presents the architecture of VerCors. The general principle is that an input program is parsed and converted into the internal AST called Common Object Language (COL). Then various passes are applied to the COL AST depending on the input language and provided flags. Finally, after applying all necessary passes, the COL AST is converted into Silver, the input language of Viper [24]. Viper reports if there are any failed assertions by translating the input into SMT and calling the Z3 SMT solver [23]. These errors are translated back by VerCors to the level of the input file. For more details on the architecture and implementation of VerCors, we refer the reader to “The VerCors Tool Set: Verification of Parallel and Concurrent Software” by Blom et al. [6].

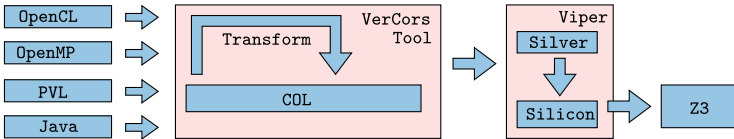


Fig. 1. The architecture of the VerCors tool.

VerCors exposes deductive logic through pre- and postconditions. These are added to the program through annotations, after which VerCors verifies the program if the program adheres to the annotations. Pre- and postconditions are sometimes transformed into assertions, but the semantics remain unchanged.

Listing 1. An implementation of a method computing the maximum of two integers.

```

1  //@ ensures (a > b ? a : b) == \result;
2  //@ signals (ArithmeticException e) a < 0 || b < 0;
3  int max(int a, int b) {
4      if (a < 0 || b < 0) { throw new ArithmeticException(); }
5      return a > b ? a : b; }

```

The VerCors pre- and postcondition syntax is inspired by JML [21]. In the example given in Listing 1, the `max` method is given a contract on line 1 that specifies that its result has to be equal to the maximum of `a` or `b`. Note how the contract is preceded with `//@`, indicating that this comment is in fact a verification annotation. The `ensures` keyword indicates this is a postcondition.

One example of a more complicated contract is the `signals` clause, which first appeared in JML [21]. It is similar to a regular `ensures` postcondition, but only holds if a certain type of exception is thrown. In Listing 1 on line 2 a `signals` clause specifies that if the method throws an `ArithmeticException` it can be assumed that the arguments are negative. Note that the `signals` clause does not impose an obligation to throw when `a < 0 || b < 0`. It *only* indicates that *if* an exception is thrown, the given exceptional postcondition holds.

3 Related Work

There are several tools that support exceptions, each with their own level of support. The following tools support separation logic: Nagini, Gillian-JS, Verifast and jStar. These other tools do not: KeY, OpenJML and Krakatoa. Table 1 summarizes the tools discussed in this section.

Nagini. Nagini fully supports exceptions in the Python language, including the Python equivalents of the statements `break`, `continue`, `return`, `try`, `catch`, and `finally`. This is done by encoding the control flow into an auxiliary state variable that indicates the type of control flow. This approach is documented in the code documentation of Nagini [10].

At first sight it seems that the Python exception model is identical to the exception model of Java. However, there is one subtle difference: Python does not allow labelled breaks. As labelled `breaks` complicate the verification of `finally` (explained in Sect. 5), the implementation strategy employed by Nagini is not directly usable for verifying exceptions in Java and would have to be extended.

Gillian-JS. Gillian-JS [13], formerly known as JaVerT [11], supports exceptions as defined in ECMAScript 5 Strict mode fully. Strict mode is a restricted version of JavaScript where pitfalls of original JavaScript are interpreted as errors.

Table 1. Related work summary

| Name | Language | Separation logic | Exceptions |
|----------------------------|------------|------------------|------------------------|
| Nagini | Python | Yes | Yes |
| Gillian-JS | JavaScript | Yes | Yes |
| Verifast | Java | Yes | Up to finally |
| jStar | Java | Yes | Trivial finally |
| KeY | Java | No | Yes |
| OpenJML | Java | No | Yes |
| Krakatoa | Java | No | Up to finally |

Through private communication with the authors of Gillian-JS we have concluded that Gillian-JS uses the inlining approach. This makes Gillian-JS susceptible to blow-up of the AST size when nested **finally** blocks occur, but the authors of Gillian-JS say they have had no problems with this in practice.

One interesting aspect of Gillian-JS is that internally it keeps track of the following four pieces of information while processing commands: the current error variable, the current return value variable, and the nearest **break** and **continue** labels. This could be simplified by using the approach presented in this work, which, if used, would only need the following two pieces of information: the current exception variable, and the nearest **try-catch-finally** block.

Verifast. Verifast [19] almost fully supports Java exceptions. This means **break**, **return**, **continue**, **throw**, **try**, and **catch** are all supported. These are encoded directly into SMT. The only language feature missing is **finally**. As mentioned in [18], the authors of Verifast are not sure how to encode **finally** clauses.

jStar. jStar [8] has some support for exceptions. Specifically, it allows use of the **try-finally** statement if it can be optimized away trivially. Otherwise jStar crashes. This optimizing is done by Soot [33], an analysis framework for Java.

Soot can parse Java bytecode into its internal representation Jimple. Then, it can apply transformations to this internal representation. Soot can also do a degree of static analysis, which allows it to remove parts of the program if it can detect that it is never executed. For example, if it can detect that the condition of an **if** statement is always true, it will remove the false branch of the **if**. This simplified Jimple code is processed by jStar for analysis.

For convenience we have included a test setup with instructions for running jStar in the package accompanying this paper [30].

KeY. KeY supports sequential Java exceptions. KeY is based on the JavaDL logic, as described in “The KeY Book” [1]. JavaDL provides axiomatic rules for dealing with exceptions, **return**, and labelled **break**. Support for **continue**

is implemented by transforming it into `break`. Within these axiomatic rules, control flow is encoded through control flow flags, as described in [32].

Steinhöfel and Wasser present the loop scopes approach that will soon replace the control flow flags approach [32]. Loop scopes reduce the number of proof obligations KeY generates in most cases when dealing with abrupt termination in loops. This is achieved by generalizing the various notions of abrupt termination into the concept of a loop scope.

OpenJML. OpenJML [7] also supports sequential Java exceptions, as well as extensive JML support for specifying the behaviour of exceptions. Steinhöfel and Wasser mention that exceptions and abrupt termination are implemented in OpenJML by encoding the control flow in `goto` [32, Sec. 6].

Krakatoa. Krakatoa [22] supports exceptions, but not `finally`. It achieves this by compiling Java exceptions into the more limited exception model of WhyML. By running the latest version of Krakatoa with `finally` in the input, we have concluded that it does not support `finally`. A test setup with instructions to check this is included in the package accompanying this paper [30].

Krakatoa takes a similar approach to this work by encoding Java exceptions into the cleaner exception model of WhyML. Also similar to our work, they use this approach to implement the abrupt termination semantics of `continue` and `break`. Surprisingly, the developers of Krakatoa seem to have missed the insight that the approach of encoding abrupt termination into exceptions can be applied to `finally`. Since Krakatoa uses an architecture based on an intermediate representation that is passed through various transformations, we expect that applying this insight could simplify the implementation of Krakatoa.

4 Semantics of Exceptions

In this section we describe the semantics of exceptions that we have implemented in VerCors. First we define how we separate error types from error causes. Then, we describe what the ideal semantics is, and why we have not implemented it. Finally, we describe the approximation semantics that we have settled on.

4.1 Errors and Sources of Errors

In Java, an exception of a subclass of `Error` is thrown when a runtime problem occurs. It is important to separate the error types from the error sources, i.e. the exception types that are thrown from the events that cause them to be thrown. We define operations that can cause an `Error` to be thrown as “sources of errors”.

For example, when allocating a new object, it can occur that the system is out of memory. In response to this, the allocation terminates abruptly, and throws an exception of type `OutOfMemoryError`. In this case, the error type is `OutOfMemoryError`. The source of the error is the system running out of memory

while allocating a new object. Some other Java `Error` types and their sources are: `OutOfMemoryError` caused by loading a new class, and `NoClassDefFoundError` if a class that needs to be loaded is absent. Note that a single error type, e.g. `ClassFormatError`, can be caused by many sources of errors.

4.2 Ideal Semantics

An ideal static analysis tool would follow the semantics outlined in the JLS to the letter. Taking this approach would result in a tool that can analyse the behaviour of a program close to its actual runtime behaviour. Unfortunately, this is not a useful approach for two reasons.

First, the annotation overhead would be enormous. This is because of `OutOfMemoryError`, which occurs when there is no more free memory. Java programs do many allocations, e.g. incrementing an `Integer` object allocates a new `Integer`. Since deductive verification requires annotating for exceptions, the ideal semantics would require every method that allocates an object to specify a contract for `OutOfMemoryError`. However, this is often a meaningless contract, because the system would crash in that case, and no recovery is possible. Therefore, formalizing error sources such as the system being out of memory in a tool will require many superfluous annotations in programs, to be specified by the user.

Second, some exceptions cannot be verified at compile time. For example, `ClassFormatError` can be thrown while loading or linking improperly formatted code. `VirtualMachineError` can be thrown because of bugs in the virtual machine. Because some errors depend on the runtime environment, static analysis tools cannot guarantee their absence. Additionally, the design rationale behind `Error` types is that regular programs do not recover from them. Paraphrasing the JLS [15, Sec. 11.1.1]: “`Error` is the superclass of all the exceptions from which ordinary programs are not ordinarily expected to recover.”.

4.3 Semantics Modulo Errors

To avoid the problems with the ideal semantics, we define a simplified view of exceptions where only a subset of the ideal exceptions semantics is included. We refer to this view of exceptions as “exceptions modulo errors”. In this view, exceptions can only come from the `throw` statement and method calls.

Formally, if VerCors does not report any problems when verifying a program it implies the following guarantee:

Definition 1 (Exception guarantee). *Any exception from a `throw` statement or a method call is handled in a surrounding `catch`, or the method declares the exception type in a `signals` or a `throws` clause. In addition, during execution the following errors will not occur:*

- *`NullPointerException` when a `null` reference is dereferenced.*
- *`ArithmeticException` when division by zero or modulo zero takes place.*
- *`ArrayIndexOutOfBoundsException` for out of bounds array accesses.*

This definition does not guarantee if an exception will be thrown at all. Therefore it is similar to the notion of partial correctness, which states that a postcondition of a program only holds *if* a program terminates at all.

In short, the exception guarantee implies that all exceptions originating from most common operations, or where the users specify them, are handled through **catch**, **signals**, or **throws**.

While the exception guarantee reduces the annotation burden on the user, it must be emphasised that this is a trade-off. In other words, the guarantee is weaker than what happens in practice. For example, some allocations may fail, but these are not modelled by the exception guarantee. Therefore, the exception guarantee will allow some bugs to go unnoticed. We leave annotation and verification of error sources for future work.

5 The finally Encoding Problem

In the previous section we have introduced the semantics that VerCors uses for reasoning about exceptions. In this section, we discuss how VerCors implements this semantics as several program transformations. Specifically, we discuss how the combination of regular control flow and exceptional control flow in **finally** clauses complicates the transformation, and how we resolve this.

Encoding abrupt termination into **goto** is straightforward if **finally** is not present. This is because the description of the semantics as given in the JLS can be interpreted literally. An overview of the transformation is as follows.

- **throw** redirects control flow to the nearest handler or exits the method.
- After a **catch** clause execution continues after the **try**.
- **break** redirects control flow to after the nearest loop.
- **return** redirects control flow to the end of a method.
- When method calls throw an exception, control flow is redirected to the nearest handler or to the end of the method.
- If **try** terminates normally execution should continue after the **try** block.

However, when **finally** is introduced, a more intricate transformation is needed. This is because contrary to all other abrupt termination primitives, at the end of a **finally** clause, it is not directly clear where to jump to.

Consider the example in Listing 2. The lines indicate how control flow would progress. The **break** statements on lines 5 and 7 both redirect control flow to the **finally** block, as it must be executed before leaving the inner **while** loop. Then, at the end of the **finally** block, control flow continues to either directly after the inner, or directly after the outer while loop. The control flow after the **finally** splits because the **break** statements are subtly different. The first **break** statement is unlabelled, which means it breaks from the most recently entered **while** loop. The second **break** statement is labelled, which means it breaks from the **while** loop that has that label.

Listing 2. Breaks can introduce ambiguous code paths.

```

1   L: while (p) {
2       while (q) {
3           try {
4               if (r) {
5                   break;
6               } else if (s) {
7                   break L;
8               }
9           } finally {
10              /* Ambiguity */
11          }
12      }
13  }
14 }
15 }

```

With the control flow explicitly drawn, reasoning about the control flow is easy. However, without the lines it is less clear what exactly must happen on line 10. If `break` was just executed, control flow needs to jump to after the `while` on line 13. If `break L` was just executed, control flow needs to jump to after the outer `while` loop on line 15. Without any further information, there is an ambiguity on line 10 which can only be resolved by knowing what kind of statement was previously executed. Hence, we argue that `finally` is non-modular in the sense that its semantics can only be determined when taking into account multiple parts of a method, and not just the `finally` clause itself.

To encode `finally` blocks, what “kind” (returning, breaking, or throwing) of control flow currently applies needs to be encoded. Furthermore, once labelled breaks are added to the language it becomes even more complicated since which *specific* loop is to be broken out of also needs to be tracked.

5.1 Candidate Encodings

Several candidate encodings for `finally` and the rest of the abrupt termination primitives are possible. We discuss the three encodings known to us next. These are: using inlining, using control flow flags, and using exceptions.

While the first and second of these encodings have appeared in some form in an implementation before this work, we have not yet seen an effort to categorize and compare the approaches.

Inlining. The first option that comes to mind is to inline all `finally` blocks in places where normally control flow would jump to the next place of interest. For example, before a throwing method call would jump to a handler, the `finally` clause could be executed by inlining it right there.

Listing 3. Before transformation.

```

try { m1();
      m2();
} finally {
    try { m3();
          m4();
    } finally { inner(); } }
    
```

Listing 4. After transformation.

```

m1(); if (exc) {
    m3(); if (exc) inner();
    m4(); if (exc) inner(); }
m2(); if (exc) {
    m3(); if (exc) inner();
    m4(); if (exc) inner(); }
    
```

Fig. 2. Transformation of inlining `finally`. `m1-4` are assumed to be throwing. Pseudo exception handling syntax is used in Listing 4, where `exc` evaluates to `true` if the previous line threw an exception.

This option is interesting because it is conceptually straightforward. It is also used in Java compilers [17, p. 3], informally showing that the approach works. The downside of this encoding is possibly exponential code duplication. Figure 2 shows a practical example of the inlining approach where this exponential duplication happens. Notice how the call to `inner` is duplicated four times. This is because the number of times the inner `finally` is duplicated is equal to the product of the number of times it must be inlined in the inner and outer `try`.

The blow-up caused by inlining was shown to be minimal for regular Java code by Stephen Freund [12]. However, for Java code containing verification annotations, it is unknown if this is also the case, as verification annotations can contain proof steps. Therefore, this cannot be assumed to be the case for verification code as well. Moreover, it is bad for the prover backend, as duplicated code might cause duplicated proof obligations, which will increase the time needed to prove the program correct. We have performed an informal experiment that shows this could be the case for VerCors. This experiment is discussed in [29].

Gillian-JS, as discussed in Sect. 3, uses the inlining approach.

Control Flow Flags. The second option is the optimized version of the first option: `finally` blocks are not inlined, but instead a flag is set whenever the mode of control flow changes. For example, when a `return` is executed, the flag is set to a constant called `MODE_RETURN`. This flag can then be queried at the end of a `finally` clause to determine where next to jump to. There should be values for each available mode of abrupt termination (i.e. `break`, `return`, `throw`), as well as a mode for every label that can be broken from.

As far as we can tell this is technically possible, but keeping track of all the labels and modes available seems error-prone. Furthermore, at the end of every `finally` clause there has to be an `if` statement determining where to jump next. In a way, this `if` statements encodes all possible origins of the `finally` block, and all possible destinations. This means the `if` statement is non-modular, as it needs information from various places in the method. This introduces unnecessary complexity and increases the chances for bugs.

Listing 5. Before transformation.

```
L: while (c) {
    ... break L; ...
}
```

Listing 6. After transformation.

```
try { while (c) {
    ... throw new L(); ...
} } catch (L e) { }
```

Fig. 3. Transformation of `break` to `throw` and `catch`.

This approach has been proposed before, but formulated differently, by Freund [12]. He proposes to encode `finally` into `goto` by transforming each `finally` into a subroutine. Before such a subroutine is called, a unique number is pushed on the stack. This unique number corresponds to the return address of the subroutine, which is recorded in a table. Even though this formulation is different than ours, the downsides of the control flow flag approach still apply.

Nagini, as discussed in Sect. 3, uses the control flow flag approach.

Exceptions. The third option is to consider abrupt termination from an exceptional point of view. When only exceptional control flow is considered, the question of where to continue at the end of a `finally` clause is simplified:

- If an exception is currently being thrown, execution should continue at the next nearest `catch` or `finally`. If there is no such clause, execution should go to the end of the method.
- Otherwise, execution continues after the `try-finally` block.

Note that the choice of where to jump after a `finally` clause becomes more local: it does not matter how many exceptions or labels are in scope. Only the next `finally` or `catch` clause needs to be known. Homogenizing control flow into the exceptional model simplifies the choice at the end of a `finally` clause.

A requirement of this encoding is the requirement for this simplification to apply: all other abrupt termination must be removed or transformed into exceptional control flow. This is extra work, but we argue that it is not difficult. An example of how `break` can be encoded as `throw` can be seen in Fig. 3.

The translation is similar for `continue` and `return`:

- For a statement `continue L`, the body of the while loop that is the target of the `continue` must be wrapped in a `try { ... } catch (ContinueL e) { }` block. The `continue` statement is replaced by `throw new ContinueL()`.
- For a statement `return`, the body of the method must be wrapped in `try { ... } catch (Return e) { }`. The `return` statement is replaced by `throw new Return()`. For return statements that return a value, the `Return` exception type thrown can be augmented with a field to store the value.

Other control-flow related statements, such as `throw`, `if`, `try`, `catch` and `while` are not transformed. Extended forms of `try-catch`, such as `try-with-resources`, can be supported by transforming the statement into

`try-finally`, as described in the JLS [15, Sec. 14.20.3.1]. `try-with-resources` is currently not supported in VerCors.

After this step, the remaining `throw` and `try-catch-finally` statements can be transformed into `goto` following the approach outlined at the beginning of Sect. 5, with two differences:

1. `throw` is converted into a `goto` to the nearest `finally` or `catch` clause. Throwing methods are handled similarly.
2. At the end of a `finally`, if the current control flow is exceptional, control flow must jump to the next nearest `catch` or `finally` clause. Otherwise, control flow must continue after the `try-finally`.

Because the exceptions approach results in a less error-prone encoding we have implemented it in VerCors. The encoding is used if `finally` is present in a method. If `finally` is not present, the basic encoding into `goto` (which was discussed at the beginning of Sect. 5) is used for a cleaner back-end output. The implementation can be found via the VerCors homepage [34].

A downside of compiling to exceptions is that information is lost, because all control flow is exceptional after the transformation. If this information is needed it can be encoded in the AST. This ensures that synthetic `try-catch` and `throw` can be discerned from authentic ones. Additionally, by adding an extra flag the current control flow can be identified as synthetic or authentic.

Another downside is that this approach is not suitable for a single-pass architecture, and only works in verifiers with multiple passes. Therefore the approach is less flexible and cannot be straightforwardly applied to all verifiers.

A verifier that uses a comparable approach is Krakatoa. We discuss the differences with our encoding in Sect. 3.

6 Evaluation

Next, we evaluate if the view of exceptions modulo errors can handle exception patterns from commercial software. We answer the following research questions:

1. What are common exception patterns that occur in commercial software? (Discussed in Sect. 6.1)
2. Can VerCors verify common exception patterns? (Discussed in Sect. 6.2)

6.1 Common Exception Patterns in Commercial Software

Methodology. To find common exception patterns that occur in commercial software, we do an informal survey of the literature through a search on Google Scholar using the keywords “java”, “exceptions”, and “usage”. We look for research that is at most 5 years old, presents a categorization of exception patterns, and considers fifteen or more Java projects.

Table 2. Exception pattern overview.

| Used in <code>catch</code> | Used in <code>finally</code> |
|---|---|
| Empty | Empty |
| Log, stack trace | Log |
| <code>if</code> , <code>while</code> , <code>switch</code> , <code>continue</code> , <code>break</code> , <code>return</code> | <code>continue</code> , <code>return</code> |
| <code>throw e</code> , <code>throw new E()</code> , <code>throw new E(e)</code> | <code>throw new E()</code> |
| Nested <code>try</code> | Nested <code>try</code> |

Results. From this search, four works are selected [3, 20, 25, 28]. We have aggregated the patterns from these works, combining them into common categories. The complete table, listing each category per paper and the elements of the categories, is included in the package accompanying this paper [30]. The aggregated categories can be seen in Table 2. Columns “Used in `catch`” and “Used in `finally`” contain patterns that are used in those clauses. “Empty” means the respective clause is used without any statements.

Discussion. While all four studies categorize the use of exceptions and `catch` clauses extensively, they do not discuss the use of `finally` thoroughly. More specifically, only Bicalho de Pádua and Purohit et al. include `finally` in their measurements [3, 28]. Kery et al. and Nakshatri et al. do not include `finally` [20, 25], because they do their measurements using the Boa tool [9], which does not support the `finally` clause. As a result, `finally` is less represented in the table, and some patterns could be missing. However, as completeness was not the goal of this evaluation, this is not a major issue.

6.2 Verification with VerCors

Methodology. To show that VerCors can verify each of the common exception patterns in Sect. 6.1, an example program containing the pattern has been created for each pattern in Table 2. This yields 19 example programs. Where relevant, we added annotations for stronger guarantees, e.g., in some programs we added `assert false` to indicate control flow cannot reach that part of the program. In other programs we added postconditions to indicate what kinds of normal and exceptional control flow are possible. All of these programs are included in the package accompanying this paper [30].

Results. VerCors verifies all of the annotated example programs. In Listing 7 we show the test program contained in the file `CatchStackTrace.java`. With regard to Table 2, the program corresponds to the “Used in `catch`” column and “stack trace” entry. Particularly, in this program there are no further assertions, as printing the stack trace does not require specific pre- or postconditions.

Listing 7. Example program `CatchStackTrace.java`.

```

1 class CatchStackTrace {
2     void m () {
3         try {
4             throw new Exception();
5         } catch (Exception e) {
6             e.printStackTrace();
7         } } }

```

Discussion. An aspect of Table 2 that was ignored in the evaluation is that exception patterns can occur simultaneously within a `catch` or `finally`. We are confident this is handled correctly. However, because the purpose of this evaluation was to determine if common patterns are verifiable, we leave the aspect of combinations of patterns for future work.

7 Discussion

We will briefly discuss backend requirements and performance.

7.1 Backend Requirements

Our approach imposes two requirements on the backend: support for `goto` and support for conditional permissions.

Goto. To encode exceptional control flow within a method, the approach presented in this work relies on `goto`. Therefore, if `goto` would not be available in the backend, our transformation to `goto` would not work.

Conditional Permissions. Exceptional postconditions result in conditional permissions. Permissions are a construct used in separation logic with permissions. A permission allows reading or writing from a data location on the heap. For a more thorough introduction to separation logic, we refer the reader to [26]. Conditional permissions are permissions that apply if a condition is met. For example, the postcondition `ensures b ==> Perm(x, write)` yields a `write` permission for `x` whenever `b` holds. Exceptional postconditions cause these kinds of permissions because they are encoded as *RuntimeException is thrown ==> P*, where *P* is an arbitrary postcondition containing permissions.

Conditional permissions are not problematic for separation logic, as they are well defined. However, they might lead to unclear or verbose specifications because permissions are only usable once certain conditions have been met.

One way to avoid conditional permissions is through an “exceptional invariant”. This is a user-defined invariant that holds both at entry and exit of a `try-catch` block, and at the end of `catch` clauses. This could simplify the handling of permissions.

7.2 Performance

During the usage of the implementation, we have not seen performance issues. However, there might be three causes of performance problems in the future: bigger encoded output, complex control flow, and conditional permissions.

Bigger Encoded Output. The size of code produced for the backend (the “encoded output”) increases when exceptions are used. This is mostly due to an increase of trivial statements. Specifically, more labels, `goto`, and `if` statements for typechecks are emitted. As no complex proof obligations are introduced, we do not think the increase in encoded output will be problematic.

Complex Control Flow. When using exceptions, many jumps are introduced that target few destinations, compared to programs without exceptions. This is because every call that can throw introduces a conditional jump. All these jumps go to either a `catch` block, `finally`, or the end of the method. This seems unavoidable, as this kind of control flow is the core of exceptions in Java. If complex control flow causes longer verification times, a different format for the output encoding can be investigated, for example continuation passing style.

Conditional Permissions. When the proposed transformation is applied, more conditional permissions are produced compared to programs that do not use exceptions. We have not seen evidence that this increases verification times. If conditional permissions become a cause of performance problems, the “exceptional invariant” mentioned in the subsection above could also help with this.

8 Conclusion

We presented an approach for supporting exceptional control flow that makes it easier to support `finally`. This is achieved by first transforming all occurrences of abrupt termination into exceptional control flow. This simplifies encoding `finally` and leads to a modular transformation that can be split up into several steps. Moreover, to avoid the annotation burden caused by exceptions as defined in the JLS, we propose a simplified view called exceptions modulo errors. This view focuses on the primary sources of exceptions, `throw` statements and `throws` clauses, and disregards exceptions that are ignored in practice, such as out of memory errors or other low-level details. Finally, we have evaluated the exceptions modulo errors semantics, and conclude that exceptions modulo errors can handle common exception patterns that appear in practice.

Future work will go in several directions. It would be useful to further validate the view of exceptions modulo errors by doing an empirical study of the catching and throwing of `Error` exceptions. This can be combined by researching how to annotate for `Error` exceptions, and what kinds of contracts would be specified in the context of `Error` exceptions. Possibly the implementation in this work can also be used to design and implement support for exceptional specifications of the standard library.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: *Deductive Software Verification - The KeY Book*. LNCS, vol. 10001. Springer, Cham (2016). <https://doi.org/10.1007/978-3-319-49812-6>
2. Amighi, A., Blom, S., Huisman, M., Zaharieva-Stojanovski, M.: The VerCors project: setting up basecamp. In: *Proceedings of the Sixth PLPV Workshop*. ACM (2012). <https://doi.org/10.1145/2103776.2103785>
3. Bicalho de Pádua, G.: *Studying and Assisting the Practice of Java and C# Exception Handling*. Masters, Concordia University, February 2018
4. Black Duck Open Hub: The Apache Hadoop Open Source Project on Open Hub: Languages Page (2018). https://www.openhub.net/p/Hadoop/analyses/latest/languages_summary
5. Black Duck Open Hub: The Apache Tomcat Open Source Project on Open Hub: Languages Page (2018). https://www.openhub.net/p/tomcat/analyses/latest/languages_summary
6. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: *iFM*, vol. 10510, pp. 102–110 (2017). https://doi.org/10.1007/978-3-319-66845-1_7
7. Cok, D.R.: OpenJML: software verification for Java 7 using JML, OpenJDK, and Eclipse. *EPTCS* (2014). <https://doi.org/10.4204/EPTCS.149.8>
8. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: *Proceedings of the 23rd ACM SIGPLAN OOPSLA Conference*. ACM (2008). <https://doi.org/10.1145/1449764.1449782>
9. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In: *2013 35th ICSE*. IEEE (2013). <https://doi.org/10.1109/icse.2013.6606588>
10. Eilers, M.: Shortened github link to code-level documentation of `get_finally_var` method (2021). <https://edu.nl/8a9qe>
11. Fragoso Santos, J., Maksimović, P., Naudžiūnienė, D., Wood, T., Gardner, P.: JaVerT: JavaScript verification toolchain. In: *Proceedings of the ACM Programming Language 2(POPL)* (2017). <https://doi.org/10.1145/3158138>
12. Freund, S.N.: The costs and benefits of Java bytecode subroutines. In: *Formal Underpinnings of Java Workshop at OOPSLA 98* (1998)
13. Gillian Team: Gillian - a multi-language platform for compositional symbolic analysis (2020). <https://gillianplatform.github.io/>
14. Gosling, J., Joy, B., Steele, G., Bracha, G.: *The Java language specification*, Java SE 7th edn. (2000)
15. Gosling, J., et al.: *The Java language specification*, Java SE 16th edn. (2021)
16. Hähnle, R., Huisman, M.: *Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools*. Springer (2019)
17. Hamilton, J., Danicic, S.: An evaluation of current java bytecode decompilers. In: *Ninth IEEE SCAM* (2009). DOI: 10.1109/SCAM.2009.24
18. Jacobs, B.: Verifast & Java’s “finally” clause (2020). <https://groups.google.com/forum/#!topic/verifast/56uhVmdERwA>
19. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: *Programming Languages and Systems*, vol. 6461. Springer (2010). https://doi.org/10.1007/978-3-642-17164-2_21

20. Kery, M.B., Le Goues, C., Myers, B.A.: Examining programmer practices for locally handling exceptions. In: Proceedings of the 13th MSR Conference. ACM (2016). <https://doi.org/10.1145/2901739.2903497>
21. Leavens, G.T., et al.: JML reference manual (2008). <https://www.cs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman.toc.html>
22. Marché, C., Paulin-Mohring, C., Urbain, X.: The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming* 58, 89-106 (2004). <https://doi.org/10.1016/j.jlap.2003.07.006>
23. de Moura, L., Bjørner, N.: Z3: an efficient smt solver. In: TACAS. Springer (2008)
24. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: VMCAI. Springer (2016)
25. Nakshatri, S., Hegde, M., Thandra, S.: Analysis of exception handling patterns in java projects: an empirical study. In: Proceedings of the 13th MSR Conference (2016). <https://doi.org/10.1145/2901739.2903499>
26. O’Hearn, P.: Separation logic. *Commun. ACM* 62 (2019). <https://doi.org/10.1145/3211968>
27. Osman, H., Chiş, A., Schaerer, J., Ghafari, M., Nierstrasz, O.: On the evolution of exception usage in Java projects. In: 2017 IEEE 24th SANER Conference (2017). <https://doi.org/10.1109/SANER.2017.7884646>
28. Purohit, P., Tokekar, V.: An investigation of exception handling practices in .NET and Java environments. *Int. J. Appl. Eng. Res.* 13, 2130–2140 (2018)
29. Rubbens, R.: Improving support for Java exceptions and inheritance in VerCors. Master’s thesis, University of Twente (2020). <https://essay.utwente.nl/81338/>
30. Rubbens, R.: Modular Transformation of Java Exceptions Modulo Errors: accompanying package (2021). <https://doi.org/10.4121/14905251>
31. Sena, D., Coelho, R., Kulesza, U., Bonifácio, R.: Understanding the exception handling strategies of Java libraries: an empirical study. In: Proceedings of the 13th MSR Conference. ACM (2016). <https://doi.org/10.1145/2901739.2901757>
32. Steinhöfel, D., Wasser, N.: A New Invariant Rule for the Analysis of Loops with Non-standard Control Flows. In: IFM, vol. 10510. Springer (2017). https://doi.org/10.1007/978-3-319-66845-1_18
33. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: a java bytecode optimization framework. CASCON First Decade High Impact Papers (2010). <https://doi.org/10.1145/1925805.1925818>
34. VerCors Team: VerCors homepage (2020). <https://vercors.ewi.utwente.nl/>