

Reasoning About Exceptional Behavior At the Level of Java Bytecode with BYTEBACK

MARCO PAGANONI and CARLO A. FURIA, Software Institute, USI Università della Svizzera italiana, Switzerland

A program’s exceptional behavior can substantially complicate its control flow, and hence accurately reasoning about the program’s correctness. On the other hand, formally verifying realistic programs is likely to involve exceptions—a ubiquitous feature in modern programming languages.

In this paper, we present a novel approach to verify the exceptional behavior of Java programs, which extends our previous work on BYTEBACK. BYTEBACK works on a program’s bytecode, while providing means to specify the intended behavior at the source-code level; this approach sets BYTEBACK apart from most state-of-the-art verifiers that target source code. To explicitly model a program’s exceptional behavior in a way that is amenable to formal reasoning, we introduce Vimp: a high-level bytecode representation that extends the Soot framework’s Jimple with verification-oriented features, thus serving as an intermediate layer between bytecode and the Boogie intermediate verification language. Working on bytecode through this intermediate layer brings flexibility and adaptability to new language versions and variants: as our experiments demonstrate, BYTEBACK can verify programs involving exceptional behavior in all versions of Java, as well as in Scala and Kotlin (two other popular JVM languages).

1 INTRODUCTION

Nearly every modern programming language supports exceptions as a mechanism to signal and handle unusual runtime conditions (so-called *exceptional behavior*) separately from the main control flow (the program’s *normal* behavior). Exceptions are usually preferable to lower-level ad hoc solutions (such as error codes and defensive programming), because deploying them does not pollute the source code’s structured control flow. However, by introducing extra, often implicit execution paths, exceptions may also complicate reasoning about all possible program behavior—and thus, ultimately, about program correctness.

In this paper, we introduce a novel approach to perform deductive verification of Java programs involving exceptional behavior. Exceptions were baked into the Java programming language since its inception, where they remain widely used [?? ?]; nevertheless, as the example of ?? demonstrates, they can be somewhat of a challenge to reason about formally. To model together normal and exceptional control flow paths, and to seamlessly support any exception-related language features, our verification approach crucially targets a program’s *bytecode* intermediate representation—instead

Authors’ address: Marco Paganoni, marco.paganoni@usi.ch; Carlo A. Furia, bugcounting.net, Software Institute, USI Università della Svizzera italiana, Lugano, Switzerland.

of the source code analyzed by state-of-the-art verifiers such as KeY [?] and OpenJML [?]. We introduced the idea of performing formal verification at the level of bytecode in previous work [?]. In this paper, we build on those results and implement support for exceptions in the BYTEBACK deductive verifier.

The key idea of our BYTEBACK approach (pictured in ??) is using JVM bytecode solely as a convenient intermediate representation; users of BYTEBACK still annotate program source code in a very similar way as if they were working with a source-level verifier. To this end, we extend the specification library introduced with BYTEBACK (called BBLib) with features to specify exceptional behavior (for example, conditions under which a method terminates normally or exceptionally) using custom Java expressions; thus, such specifications remain available in bytecode after compiling an annotated program using a standard Java compiler. BYTEBACK analyzes the bytecode and encodes the program’s semantics, its specification, and other information necessary for verification into Boogie [?]—a widely-used intermediate language for verification; then, verifying the Boogie translation is equivalent to verifying the original Java program against its specification.

As we demonstrate with experiments in ??, performing verification on bytecode offers several advantages: *i) Robustness* to source-language changes: while Java evolves rapidly, frequently introducing new features (also for exceptional behavior), bytecode is generally stable; thus, our verification technique continues to work with the latest Java versions. *ii) Multi-language* support: BYTEBACK and its BBLib specification library are designed so that they can be applied, in principle, to specify programs in any language that is bytecode-compatible; while the bulk of our examples are in Java, we will demonstrate verifying exceptional behavior in Scala and Kotlin—two modern languages for the JVM. *iii) Flexibility* of modeling: since exceptional behavior becomes explicit in bytecode, the BYTEBACK approach extensively and seamlessly deals with any intricate exceptional behavior (such as implicit or suppressed exceptions). As we further illustrate in ??, reasoning about exceptional behavior of Java (and other JVM languages) is already fully supported also at the source-code level. In ?? we discuss how the aforementioned advantages of targeting bytecode are largely complementary to the advantages of working at the source level—which is the standard approach taken by most Java verifiers.

1.1 Contributions and Positioning

In summary, the paper makes the following contributions: *i)* Specification features to specify exceptional behavior of JVM languages. *ii)* A verification technique that encodes bytecode exceptional behavior into Boogie. *iii)* An implementation of the specification features and the verification technique that extend the BBLib library and BYTEBACK verifier. *iv)* Vimp: a high-level bytecode format suitable to reason about functional correctness, built on top of Soot’s Jimple format [? ?]. *v)* An experimental evaluation with 60 programs involving exceptional behavior in Java, Scala, and

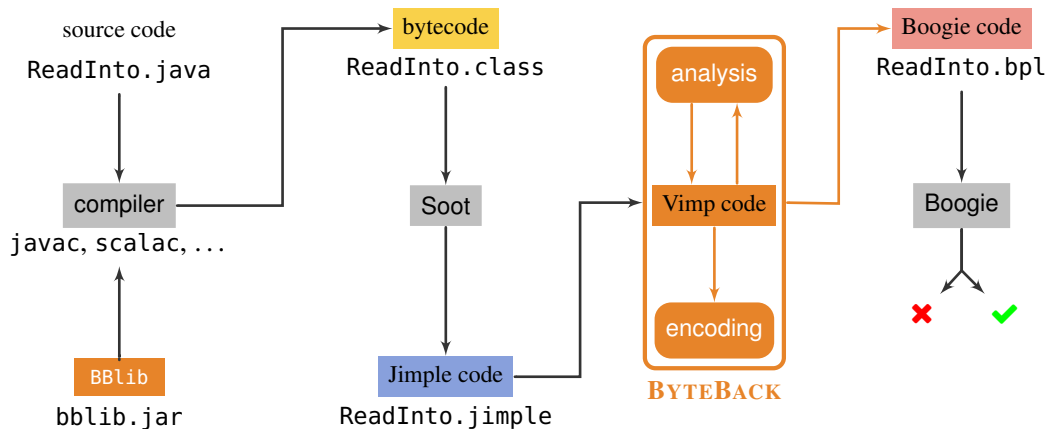


Fig. 1. An overview of BYTEBACK’s verification workflow.

Kotlin. *vi*) For reproducibility, BYTEBACK and all experimental artifacts are available in a replication package [?]. While we build upon BBLib and BYTEBACK, introduced in previous work of ours [?], this paper’s contributions substantially extend them with support for exceptional behavior, as well as other Java related features (see ??). For simplicity, henceforth “BBLib” and “BYTEBACK” denote their current versions, equipped with the novel contributions described in the rest of the paper.

1.2 Current State of BYTEBACK

This is the third publication on BYTEBACK to date; here is a summary of the new contributions in each of them.

- “Verifying Functional Correctness Properties At the Level of Java Bytecode”, presented at FM 2023 [?], introduced the BYTEBACK technique, the BBLib annotation library, and the first release of the tool. Just like later versions of BYTEBACK, FM 2023’s BYTEBACK uses Soot to analyze bytecode (precisely, Soot’s Grimp representation) and Boogie as verification backend.¹ In contrast, FM 2023’s BYTEBACK lacks support for specifying or reasoning about exceptional behavior.
- “Reasoning About Exceptional Behavior At the Level of Java Bytecode”, presented at iFM 2023 [?], revised and extended the BYTEBACK technique, as well as the BBLib annotation library, to support specifying and reasoning about exceptional behavior. To this end, it also introduces Vimp: a custom high-level bytecode representation that simplifies expressing

¹All versions of BYTEBACK to date use recent versions of Soot and Boogie without modifications: Soot is used through its API, whereas Boogie is used as a command-line tool.

some of the new transformations needed to analyze exceptional behavior. iFM 2023’s BYTEBACK still uses Soot to analyze bytecode and Boogie as verification backend.

- The present paper is an extended version of the iFM 2023 publication [?], which *i)* describes in greater detail BYTEBACK’s technique for reasoning about exceptional behavior; *ii)* introduces new exception-related features of BYTEBACK, such as the handling of more kinds of implicitly thrown exceptions; *iii)* includes a more extensive experimental evaluation with several more benchmark programs in Java, Scala, and Kotlin; *iv)* is accompanied by a new release of the BYTEBACK tool, which is made available in a new replication package [?]. While this paper’s release of BYTEBACK benefits from several minor improvements in the design and fixes a few minor bugs, its architecture is essentially the same as iFM 2023’s BYTEBACK’s.

2 MOTIVATING EXAMPLE

Exceptions can significantly complicate the control flow of even seemingly simple code; consequently, correctly reasoning about exceptional behavior can be challenging even for a language like Java—whose exception-handling features have not changed significantly since the language’s origins.

To demonstrate, consider `??`’s method `into`, which inputs a reference `r` to a `Resource` object and an integer array `a`, and copies values from `r` into `a`—until either `a` is filled up or there are no more values in `r` to read. `??` shows the key features of class `Resource`—which implements Java’s `AutoCloseable` interface, and hence can be used similarly to most standard I/O classes. Method `into`’s implementation uses a `try-with-resources` block to ensure that `r` is closed whenever the method terminates—normally or exceptionally. The `while` loop terminates as soon as any of the following conditions holds: *i)* `r` is `null`; *ii)* `i` reaches `a.length` (array `a` is full), and hence the assignment to `a[i]` throws an `IndexOutOfBoundsException`; *iii)* `r` has no more elements, and hence `r.read()` throws a `NoSuchElementException`. Correspondingly, in case `??` method `into` returns with an (propagated) exception; in cases `??` and `??` the `IndexOutOfBoundsException` or `NoSuchElementException` exceptions that are thrown are immediately caught by the `catch` block that *suppresses* them with a `return`—so that `into` returns normally.

`??`’s annotations, which use a simplified syntax for `BBLib`—BYTEBACK’s specification library—specify part of `into`’s expected behavior. Precondition `@Require` expresses the constraint that object `r` must be `null` or open (not closed) for `into` to work as intended. Annotation `@Raise` declares that `into` terminates with a `NullPointerException` exception if `r` is `null`; conversely, `@Return` says that `into` returns *normally* if `a` and `r` are not `null`. Finally, postcondition `@Ensure` says that, if it’s not `null`, `r` will be closed when `into` terminates—regardless of whether it does so normally or exceptionally.

<pre> 1 @Require(r = null ∨ ¬r.closed) 2 @Raise(NullPointerException, r = null) 3 @Return(a ≠ null ∧ r ≠ null) 4 @Ensure(r = null ∨ r.closed) 5 public static void into(Resource r, int[] a) { 6 try (r) { 7 int i = 0; 8 while (true) { 9 invariant(0 ≤ i ≤ a.length); 10 invariant(r = null ∨ ¬r.closed); 11 a[i] = r.read(); ++i; 12 } 13 } catch (IndexOutOfBoundsException 14 NoSuchElementException e) 15 { return; } 16 } 17 </pre>	<pre> class Resource implements AutoCloseable { public boolean closed; public boolean hasNext; @Raise(IllegalStateException, closed) @Raise(NoSuchElementException, ¬hasNext) @Return(¬closed ∧ hasNext) public int read() { // ... } // ... } </pre>
---	--

(a) Method `into` copies `r`'s content into array `a`. It is annotated with normal and exceptional pre- and postconditions using a simplified BLib syntax.

(b) An outline of class `Resource`'s interface with Boolean attributes `closed` and `hasNext`, and method `read`, annotated using a simplified BLib syntax.

Fig. 2. Annotated Java method `into` and class `Resource`, which demonstrate some pitfalls of specifying and reasoning about exceptional behavior.

The combination of language features and numerous forking control-flow paths complicate reasoning about—and even specifying—`into`'s behavior. While exception handling has been part of Java since version 1, try-with-resources and multi-catch (both used in ??) were introduced in Java 7; thus, even a state-of-the-art verifier like KeY [?] lacks support for them. OpenJML [?] can reason about all exception features up to Java 7; however, the try-with-resources using an existing **final** variable became available only in Java 9.² Furthermore, OpenJML implicitly checks that verified code does not throw any implicit exceptions (such as `NullPointerException` or `IndexOutOfBoundsException`)—thus disallowing code such as ??'s, where implicitly throwing exceptions is part of the expected behavior. These observations are best thought of as design choices—rather than limitations—of these powerful Java verification tools: after all, features such as multi-catch are syntactic sugar that makes programs more concise but does not affect expressiveness; and propagating uncaught implicit exceptions can be considered an anti-pattern³ [?]. However, they also speak volumes to the difficulty of fully supporting all cases of exceptional behavior in a feature-laden language like Java.

² <https://jcp.org/en/jsr/detail?id=334>

³ <https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html>

As we detail in the rest of the paper, BYTEBACK’s approach offers advantages in such scenarios. Crucially, the implicit control flow of exception-handling code becomes explicit when compiled to bytecode, which eases analyzing it consistently and disentangling the various specification elements. For instance, the **while** loop’s several exceptional exit points become apparent in `into`’s bytecode translation, and BYTEBACK can check that the declared invariant holds in all of them, and that postcondition `@Ensure` holds in all matching method return points. Furthermore, bytecode is more stable than Java—thus, a verifier like BYTEBACK can seamlessly handle old, as well as recent, Java versions. Thanks to these capabilities, BYTEBACK can verify the behavior of ??’s example.

3 SPECIFYING AND VERIFYING EXCEPTIONAL BEHAVIOR

This section describes the new features of BYTEBACK to specify and verify exceptional behavior. ?? shows BYTEBACK’s workflow, which we revisited to support these new features.

Users of BYTEBACK—just like with every deductive verifier—have to annotate the source code to be verified with a specification and other annotations. To this end, BYTEBACK offers `BBlib`: an annotation library that is usable with any language that is bytecode compatible. ?? describes the new `BBlib` features to specify exceptional behavior. Then, users compile the annotated source code with the language’s bytecode compiler. BYTEBACK relies on the Soot static analysis framework to analyze bytecode; precisely, Soot offers `Jimple`: a higher-level alternative bytecode representation. BYTEBACK processes `Jimple` and translates it to `Vimp`: a verification-oriented extension of `Jimple` that we introduced in the latest BYTEBACK version and is described in ?. As we discuss in Sections ?? and and ??, BYTEBACK transforms `Jimple` to `Vimp` in steps, each taking care of a different aspect (expressions, types, control flow, and so on). Once the transformation is complete, BYTEBACK encodes the `Vimp` program into the Boogie intermediate verification language [?]; as we discuss in in ??, the Boogie encoding is mostly straightforward thanks to `Vimp`’s custom design. Finally, the Boogie tool verifies the generated Boogie program, and reports success or any verification failures (which can be manually traced back to source-code specifications that could not be verified).

3.1 Specifying Exceptional Behavior

Users of BYTEBACK add behavioral specifications to a program’s source using `BBlib`: BYTEBACK’s standalone Java library, offering annotation tags and static methods suitable to specify functional behavior. Since `BBlib` uses only basic language constructs, it is compatible with most JVM languages (as we demonstrate in ??); and all the information added as `BBlib` annotations is preserved at the bytecode level. This section first summarizes the core characteristics of `BBlib` (to make the paper self contained), and then describes the features we introduced to specify exceptional behavior.

3.1.1 Specification Expressions. Expressions used in BYTEBACK specifications must be *aggregable*, that is pure (they can be evaluated without side effects) and branchless (they can be evaluated without branching instructions). These are common requirements to ensure that specification expressions are well-formed [?]—hence, equivalently expressible as purely *logic expressions*. Correspondingly, BBLib forbids impure expressions, and offers aggregable replacements for the many Java operators that introduce branches in the bytecode, such as the standard Boolean operators (&&, ||, ...) and comparison operators (==, <, ...). ?? shows several of BBLib’s aggregable operators, including some that have no immediately equivalent Java expression (such as the quantifiers). Using only BBLib’s operators in a specification ensures that it remains in a form that BYTEBACK can process after the source program has been compiled to bytecode [?]. Thus, BBLib operators map to Vimp logic operators (??), which directly translate to Boogie operators with matching semantics (??).

	IN JAVA/LOGIC	IN BBLib
comparison	$x < y, x \leq y, x == y$ $x != y, x \geq y, x > y$	<code>lt(x, y), lte(x, y), eq(x, y)</code> <code>neq(x, y), gte(x, y), gt(x, y)</code>
conditionals	$c ? t : e$	<code>conditional(c, t, e)</code>
propositional	$\neg a, a \ \&\& \ b, a \ \ b, a \implies b$	<code>not(a), a & b, a b, implies(a, b)</code>
quantifiers	$\forall x: T \bullet P(x)$ $\exists x: T \bullet P(x)$	<code>T x = Binding.T(); forall(x, P(x))</code> <code>T x = Binding.T(); exists(x, P(x))</code>

Table 1. BBLib’s aggregable operators, used in specification expressions instead of Java’s impure or branching operators.

3.1.2 Method Specifications. To specify the input/output behavior of methods, BBLib offers annotations `@Require` and `@Ensure` to express a method’s pre- and postconditions. For example,

`@Require(p) @Ensure(q) t m(args)`

specifies that p and q are method m ’s pre- and postcondition; both p and q denote names of so-called *behaviors*, which are methods marked with annotation `@Behavior`. As part of the verification process, BYTEBACK checks that every such behavior b is well-formed: *i)* b returns a **boolean**; *ii)* b ’s signature is the same as m ’s or, if b is used in m ’s postcondition and m ’s return type is not **void**, it also includes an additional argument that denotes m ’s returned value; *iii)* b ’s body returns a single aggregable expression; *iv)* if b ’s body calls other methods, they must also be aggregable.

For simplicity of presentation, we henceforth abuse the notation and use an identifier to denote a behavior’s *name*, the *declaration* of the method, and the *expression* that the behavior’s body returns. Consider, for example, ??, showing how one would express in actual BBLib syntax ??’s

```

@Require("r_is_null_or_open")
@Ensure("r_is_null_or_closed")
// ...
public static void into(Resource r, int[] a)

@Behavior
public static boolean r_is_null_or_open(Resource r, int[] a)
{ return eq(r, null) | not(r.isClosed); }

@Behavior
public static boolean r_is_null_or_closed(final Resource r, int[] a)
{ return eq(r, null) | r.isClosed; }

```

Fig. 3. Pre- and postconditions of ??’s method `into` in BBLib syntax.

```

class Counter {

    public int count = 0;

    @Ensure("increased_by_one")
    public void increment()
    { count = count + 1; }

    @TwoState @Behavior
    public boolean increased_by_one()
    { return eq(count, old(count) + 1) }

}

```

Fig. 4. A postcondition using `old` to refer to the method’s pre-state.

method `into`’s pre- and postcondition. In this example, we can use `r_is_null_or_open` to refer to method `r_is_null_or_open`, to the argument `"r_is_null_or_open"` of `into`’s `@Require` annotation, or to the behavior’s returned expression `eq(r, null) | not(r.isClosed)`—which is `into`’s actual precondition.

3.1.3 Two-state Postconditions. A method’s postcondition is, in general, a predicate over two states: the *pre-state* (when the method begins execution) and the *post-state* (when the method terminates). In BBLib, a behavior q used as postcondition may only refer to the pre-state if it is equipped with annotation `@TwoState`; in this case, q ’s body may include expressions `old(e)` to refer to any expression e ’s value in the pre-state. Accordingly, BYTEBACK checks that two-state

behaviors (i.e., behaviors annotated with `@TwoState`) are only used in postconditions, and are only called by other two-state behaviors (including recursively). `??` shows how one would express in actual BBLib syntax the postcondition of a method `increment` that increments attribute `count` by one: `increased_by_one` is a two-state behavior precisely expressing `increment`'s postcondition that `count`'s value in the post-state equals `count`'s value in the pre-state plus one.

3.1.4 Exceptional Postconditions. Predicate methods specified with `@Ensure` may refer to a method's post-state regardless of whether the method terminated normally or with an exception; for example, predicate `x_eq_y` in `??` says that attribute `x` always equals argument `y` when method `m` terminates. If the postcondition `q` of a method `m` is also equipped with annotation `@Exceptional`, `q`'s body may use operator `thrown()` to refer to the thrown exception object. Precisely, `isVoid(thrown())` holds if and only if `m` terminates normally; if `m` terminates with an exception, `thrown()` refers to the exception object. `??`'s methods `x_pos` and `y_neg` demonstrate using these features of BBLib: `x_pos` states that if attribute `x > 0` when `m` terminates, then it must have terminated with a `PosXExc` exception; `y_neg` states that if `m` is called with argument `y ≤ 0`, then it terminates normally.

```
public int x = 0;

@Ensure("x_eq_y") // x = y when m terminates normally or exceptionally
@Ensure("x_pos")  // if x > 0 then m throws an exception
@Ensure("y_neg")  // if y ≤ 0 then m terminates normally
public void m(int y) {
    x = y;
    if (x > 0)
        throw new PosXExc();
}

@Behavior
public boolean x_eq_y(int y)
{ return eq(x, y); }

@Exceptional @Behavior
public boolean x_pos(int y)
{ return implies(gt(x, 0), thrown() instanceof PosXExc); }

@Exceptional @Behavior
public boolean y_neg(int y)
{ return implies(lte(y, 0), isVoid(thrown())); }
```

Fig. 5. Examples of exceptional postconditions in BBLib.

3.1.5 Shorthands. For convenience, BBLib offers annotation shorthands `@Raise` and `@Return` to specify when a method terminates exceptionally or normally. `??` shows the semantics of these shorthands by translating them into equivalent postconditions. The when arguments of `@Raise` and `@Return` implicitly refer to a method’s pre-state through the `old` expression, since it is common to relate a method’s exceptional behavior to its inputs. Thus, `??`’s postcondition `y_neg` is equivalent to `@Return(lte(y, 0))`, since `m` does not change `y`’s value; conversely, `@Raise(PosXExc.class, gt(x, 0))` is *not* equivalent to `x_pos` because `m` reassigns `x` to `y`’s value upon executing.

SHORTHAND	EQUIVALENT POSTCONDITION
<code>@Raise(exception = E.class, when = p)</code>	<code>@Ensure(implies(old(p), thrown() instanceof E))</code>
<code>@Raise(when = p)</code>	<code>@Ensure(implies(old(p), not(isVoid(thrown()))))</code>
<code>@Raise(exception = E.class)</code>	<code>@Raise(exception = E.class, when = true)</code>
<code>@Raise</code>	<code>@Raise(when = true)</code>
<code>@Return(when = p)</code>	<code>@Ensure(implies(old(p), isVoid(thrown())))</code>
<code>@Return</code>	<code>@Return(when = true)</code>

Table 2. BBLib’s `@Raise` and `@Return` and annotation shorthands.

3.1.6 Intermediate Specification. BBLib also supports the usual intra-method specification elements: assertions, assumptions, and loop invariants. Given an aggregable expression `e`, `assertion(e)` specifies that `e` must hold whenever execution reaches it; `assumption(e)` restricts verification from this point on to only executions where `e` holds; and `invariant(e)` declares that `e` is an invariant of the loop within whose body it is declared. As we have seen in `??`’s running example, loop invariants hold, in particular, at all exit points of a loop—including exceptional ones.

3.2 The Vimp Intermediate Representation

In our previous work [?], BYTEBACK works directly on Grimp—a high-level bytecode representation provided by the Soot static analysis framework [?]. In this paper, we target Jimple, a different high-level bytecode representation provided by Soot, whose three-address-code form is more germane to analyzing exceptional behavior. In a three-address code, each bytecode instruction includes a single elementary operation; therefore, the control flow of a program can be modified by simply adding new instructions at any suitable point. In contrast, if an instruction may include multiple operations (for instance, a compound expression), modifying the control flow between those operations would not be as straightforward, and it would require an additional normalization step. Since supporting exceptional behavior crucially relies on modifying the bytecode control flow to make exceptional behavior explicit, a three-address-code form—where such modifications are simpler to implement—is a more effective target for our purposes.

Both Grimp and Jimple conveniently retain information such as types and expressions, which eases BYTEBACK’s encoding of the program under verification into Boogie [?]. However, Grimp and Jimple remain close to bytecode; hence, they represent well executable instructions, but lack support for encoding logic expressions and specification constructs. These limitations become especially inconvenient when reasoning about exceptional behavior, which often involves logic conditions that depend on the types and values of exceptional objects. Rather than reconstructing this information during the translation from Jimple (or Grimp) to Boogie, we found it more effective to extend Soot’s bytecode representations into Vimp, which fully supports logic and specification expressions.

Our bespoke Vimp bytecode representation can encode all the information relevant for verification. This brings several advantages: *i*) it decouples the input program’s static analysis from the generation of Boogie code, achieving more flexibility at either ends of the toolchain; *ii*) it makes the generation of Boogie code straightforward (mostly one-to-one); *iii*) BYTEBACK’s transformation from Jimple to Vimp becomes naturally *modular*: it composes several simpler transformations, each taking care of a different aspect and incorporating a different kind of information. The rest of this section presents Vimp’s key features, and how they are used by BYTEBACK’s Jimple-to-Vimp transformation \mathcal{V} .⁴

Transformation \mathcal{V} combines, as detailed in ??, the following feature-specific transformations: \mathcal{V}_{exc} makes the exceptional control flow explicit; $\mathcal{V}_{\text{inst}}$ translates Jimple instructions (by applying transformation \mathcal{V}_{exp} to expressions within instructions, \mathcal{V}_{agg} to aggregate Jimple expressions into compound Vimp expressions, and $\mathcal{V}_{\text{type}}$ to infer types); $\mathcal{V}_{\text{loop}}$ handles loop invariants. In the following subsections, we describe the most significant aspects of these transformations—focusing on those that are non-standard and are especially relevant to reasoning about exceptions.

3.2.1 Expected Types. Transformation $\mathcal{V}_{\text{type}}$ reconstructs the *expected type* of expressions when translating them to Vimp. An expression e ’s expected type depends on the context where e is used; in general, it may differ from e ’s type in Jimple, as we explain in the following paragraphs.

3.2.2 Boolean Types. Jimple’s built-in type resolver [?] may assign type `int` also to Boolean values and expressions; this is consistent with how bytecode encodes Boolean values as integers: 1 for `true` and 0 for `false`. This entails that Jimple code may assign a Boolean value to an integer variable, and vice versa. This complicates encoding bytecode Boolean expressions into Boogie, where the `boolean` and `int` types are strictly separate.

To address this issue, Vimp’s transformation $\mathcal{V}_{\text{type}}$ strictly distinguishes between `boolean` and `int` types. Since Jimple’s typing rules correctly identify Boolean types in declarations and signatures, we only need to define custom typing rules for Boolean expressions as shown in ?. Since logic

⁴In the following, we occasionally take some liberties with Jimple and Vimp code, using a readable syntax that mixes bytecode instruction and Java statement syntax; for example, $m()$ represents an invocation of method m that corresponds to a suitable variant of bytecode’s `invoke`.

$$\begin{array}{c}
\frac{\mathcal{V}_{\text{type}}(e_1) = \text{int} \quad \mathcal{V}_{\text{type}}(e_2) = \text{int}}{\mathcal{V}_{\text{type}}(e_1 \bowtie e_2) = \text{boolean}} \quad (1) \qquad \frac{\mathcal{V}_{\text{type}}(e_1) = \text{boolean} \quad \mathcal{V}_{\text{type}}(e_2) = \text{boolean}}{\mathcal{V}_{\text{type}}(e_1 \star e_2) = \text{boolean}} \quad (2) \\
\frac{\mathcal{V}_{\text{type}}(e_1) = \text{int} \quad \mathcal{V}_{\text{type}}(e_2) = \text{boolean}}{\mathcal{V}_{\text{type}}(e_1 \star e_2) = \text{int}} \quad (3) \qquad \frac{\mathcal{V}_{\text{type}}(e_1) = \text{boolean} \quad \mathcal{V}_{\text{type}}(e_2) = \text{int}}{\mathcal{V}_{\text{type}}(e_1 \star e_2) = \text{int}} \quad (4)
\end{array}$$

(a) BYTEBACK’s inference rules for the `boolean` type in expressions. \star denotes any Boolean (bitwise) logic connective, and \bowtie any integer comparison operator.

$$\begin{array}{c}
\frac{\{ \dots \text{boolean } v; \\ \dots v := e \dots \}}{\mathcal{V}_{\text{type}}(e) = \text{boolean}} \quad (5) \qquad \frac{\text{if } (e) \text{ goto } L}{\mathcal{V}_{\text{type}}(e) = \text{boolean}} \quad (6) \\
\frac{\text{boolean } m(a_1, \dots, a_m) \\ \{ \dots \text{return } e \dots \}}{\mathcal{V}_{\text{type}}(e) = \text{boolean}} \quad (7) \qquad \frac{\text{R } m(\dots \text{boolean } a \dots) \\ \dots m(\dots e \dots) \dots}{\mathcal{V}_{\text{type}}(e) = \text{boolean}} \quad (8)
\end{array}$$

(b) BYTEBACK’s inference rules for the `boolean` type in statements. Each rule’s premise shows a different context for an expression e whose type is inferred as `boolean`.

Fig. 6. How BYTEBACK handles the `boolean` type.

operations in bytecode may mix integers and Booleans, $\mathcal{V}_{\text{type}}$ assigns the more general `int` type to such expressions (rules (??) and (??) in ??). ?? shows how $\mathcal{V}_{\text{type}}$ also assigns the `boolean` type to expressions e that are used as Boolean conditions or according to their declared Boolean type: rule (??) applies to the right-hand side e of an assignment to any variable v declared as `boolean`; rule (??) applies to the condition e of a branching instruction; rule (??) applies to value e returned by a method m whose return type is `boolean`; rule (??) applies to a call to a method whose formal argument a is `boolean`.

3.2.3 Expressions. Transformation \mathcal{V}_{exp} translates expressions e according to their expected type $\mathcal{V}_{\text{type}}(e)$. For example, it uses the usual operators \neg , \wedge , \vee , \implies , and constants `true` and `false` to translate expressions involving Booleans (in contrast to bytecode’s using integer operators for Boolean operations, such as the unary minus- for “not”). \mathcal{V}_{exp} also adds explicit widening cast conversions to the operands of an expression whose expected type differs from the operator’s required type. A recurring example are “mixed” Boolean and integer expressions, such as a comparison `b = 1` where b is a Boolean variable (hence, $\mathcal{V}_{\text{type}}(b) = \text{boolean}$): according to rule (??), $\mathcal{V}_{\text{type}}(b = 1)$ is `int`, but there is no comparison operator that works to compare a Boolean and an integer; in this case, the translation $\mathcal{V}_{\text{exp}}(b = 1)$ of `b = 1` is `((int) b) = 1`—casting b to `int` so that Vimp’s standard integer comparison operator `=` can be used.

3.2.4 Quantified Expressions. Transformation $\mathcal{V}_{\text{expr}}$ also identifies quantified expressions and renders them using Vimpr’s quantifier syntax. As shown in ??, a quantified variable x of type T must be declared as $x = \text{Binding}.T()$ using `BBlib`’s `Binding` factory. Then, this is how $\mathcal{V}_{\text{expr}}$ translates a quantified expression over x whose scope is predicate $P(x)$ —after aggregating $P(x)$ as discussed in ??.

$$\mathcal{V}_{\text{exp}}(\text{Contract.forall}(x, P(x))) = \forall \mathcal{V}_{\text{type}}(x) x :: \mathcal{V}_{\text{exp}}(\mathcal{V}_{\text{agg}}(P(x)))$$

$$\mathcal{V}_{\text{exp}}(\text{Contract.exists}(x, P(x))) = \exists \mathcal{V}_{\text{type}}(x) x :: \mathcal{V}_{\text{exp}}(\mathcal{V}_{\text{agg}}(P(x)))$$

3.2.5 Expression Aggregation. Transformation \mathcal{V}_{agg} aggregates specification expressions (??), so that each corresponds to a single Jimple pure and branchless expression, which \mathcal{V}_{exp} can easily translate to Vimpr. In a nutshell, $\mathcal{V}_{\text{agg}}(e)$ takes the slice of Jimple statements that e depends on, converts it into static-single assignment form, and then recursively replaces each variable’s single usage with its unique definition. For example, consider ??’s loop invariant $0 \leq i \leq a.\text{length}$; compilation into bytecode breaks this expression down into three assignments that incrementally compute the various components of the specification expression and store into local variables, followed by a call to `BBlib` function `invariant` with the last variable as argument: x stores the value of $0 \leq i$, y the value of $i \leq a.\text{length}$, and z the value of $x \wedge y$. In order to translate expression z in invariant z , `BYTEBACK` first aggregates the three assignments back into a single Vimpr expression as shown in ??.

SPECIFICATION	SSA-FORM SLICE	AGGREGATED
invariant z	$x := \text{lte}(0, i)$ $y := \text{lte}(i, a.\text{length})$ $z := x \ \& \ y$ invariant z	invariant $\text{lte}(0, i) \ \& \ \text{lte}(i, a.\text{length})$

Table 3. Aggregating the specification expression that encodes one of ??’s loop invariants.

3.2.6 Assertion Instructions. Vimpr includes instructions `assert`, `assume`, and `invariant`, which transformation \mathcal{V}_{ins} introduces for each corresponding instance of `BBlib` assertions, assumptions, and loop invariants. Transformation $\mathcal{V}_{\text{loop}}$ relies on Soot’s loop analysis capabilities to identify loops in Vimpr’s unstructured control flow; then, it expresses their invariants by means of assertions and assumptions. As shown in ??, $\mathcal{V}_{\text{loop}}$ checks that the invariant holds upon loop entry (label head), at the end of each iteration (head again), and at every exit point (label exit).

<pre> k = 0; while (k < 10) { invariant(lte(k, 10) & lte(k, X)); k++; if (k ≥ X) break; } return k; </pre>	<pre> k := 0; head: if k ≥ 10 goto exit; invariant k ≤ 10 ∧ k ≤ X; k := k + 1; if k ≥ X goto exit; back: goto head; exit: return k; </pre>	<pre> k := 0; head: assert k ≤ 10 ∧ k ≤ X; if k ≥ 10 goto exit; assume k ≤ 10 ∧ k ≤ X; k := k + 1; if k ≥ X goto exit; back: goto head; exit: assert k ≤ 10 ∧ k ≤ X; return k; </pre>
---	--	---

Fig. 7. A loop in Java (left), its unstructured representation in Vimp (middle), and the transformation $\mathcal{V}_{\text{loop}}$ of its invariant into assertions and assumptions (right). The loop invariant expression is aggregated as described in ??.

3.3 Modeling Exceptional Control Flow

Bytecode stores a block’s exceptional behavior in a data structure called the *exception table*.⁵ Soot represents each table entry as a *trap*, which renders a try-catch block in Jimple bytecode. Precisely, a trap t is defined by: *i*) a block of instructions B_t that may throw exceptions; *ii*) the type E_t of the handled exceptions; *iii*) a label h_t to the handler instructions (which terminates with a jump back to the end of B_t). When executing B_t throws an exception whose type conforms to E_t , control jumps to h_t . At the beginning of the handler code, Jimple introduces $e := @caught$, which stores into a local variable e of the handler a reference $@caught$ to the thrown exception object. ?? shows an example of try-catch block in Java (left) and the corresponding trap in Jimple (middle): ℓ_1, \dots, ℓ_5 is the instruction block, E is the exception type, and *handler* is handler’s entry label. The rest of this section describes BYTEBACK’s transformation \mathcal{V}_{exc} , which transforms the implicit exceptional control flow of Jimple traps into explicit control flow in Vimp. Such a transformation is necessary since BYTEBACK uses Boogie as backend: the Boogie verification language only includes minimal imperative programming features, without any support for exceptional control flow; BYTEBACK makes explicit the exceptional control flow of the program under verification, and encodes this explicit exceptional behavior using Boogie’s available features.

3.3.1 Explicit Exceptional Control-Flow. Jimple’s variable $@caught$ is called $@thrown$ in Vimp. While $@caught$ is read-only in Jimple—where it only refers to the currently handled exception— $@thrown$ can be assigned to in Vimp. This is how BYTEBACK makes exceptional control flow

⁵ <https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.10>

<pre> try { x = o.size(); if (x = 0) { throw new E(); } } catch (E e) { x = 1; } </pre>	<pre> ℓ₁: x := o.size(); ℓ₂: if x != 0 goto ℓ₅; ℓ₃: e := new E(); ℓ₄: throw e; ℓ₅: goto ℓ₆; handler: e := @caught; x := 1; ℓ₆: ... </pre>	<pre> ℓ₁: x := o.size(); if @thrown = void goto ℓ₂; V_{exc}(throw @thrown); ℓ₂: if x ≠ 0 goto ℓ₅; ℓ₃: e := new E(); ℓ₄: @thrown := e; if ¬(@thrown instanceof E) goto ℓ₅; goto hE; ℓ₅: goto ℓ₆; handler: e := @thrown; @thrown := void; x := 1; ℓ₆: ... </pre>
---	---	--

Fig. 8. A try-catch block in Java (left), its unstructured representation as a trap in Jimple (middle, empty lines are for readability), and its transformation \mathcal{V}_{exc} in Vimp with explicit exceptional control flow (right).

explicit: assigning to **@thrown** an exception object e signals that e has been thrown; and setting **@thrown** to **void** marks the current execution as normal. Thus, BYTEBACK’s Vimp encoding sets **@thrown := void** at the beginning of a program’s execution, and then manipulates the special variable **@thrown** to reflect the bytecode semantics of exceptions as we outline in the following. With this approach, the Vimp encoding of a **try** block simply results from encoding each of the block’s instructions explicitly according to their potentially exceptional behavior.

3.3.2 Throw Instructions. Transformation \mathcal{V}_{exc} desugars **throw** instructions into explicit assignments to **@thrown** and jumps to the suitable handlers. A **throw** e instruction within the scope of n traps t_1, \dots, t_n —catching and handling exceptions of types E_1, \dots, E_n with handlers at labels

h_1, \dots, h_n —is transformed into:

$$\mathcal{V}_{\text{exc}}(\text{throw } e) = \left(\begin{array}{l} \text{@thrown} := e; \\ \text{if } \neg(\text{@thrown instanceof } E_1) \text{ goto } \underline{\text{skip}}_1; \\ \text{goto } h_1; \\ \underline{\text{skip}}_1: \text{if } \neg(\text{@thrown instanceof } E_2) \text{ goto } \underline{\text{skip}}_2; \\ \text{goto } h_2; \\ \underline{\text{skip}}_2: \text{if } \neg(\text{@thrown instanceof } E_3) \text{ goto } \underline{\text{skip}}_3; \\ \vdots \\ \underline{\text{skip}}_{n-1}: \text{if } \neg(\text{@thrown instanceof } E_n) \text{ goto } \underline{\text{skip}}_n; \\ \text{goto } h_n; \\ \underline{\text{skip}}_n: \text{return}; // \text{propagate exception to caller} \end{array} \right)$$

The assignment to **@thrown** stores a reference to the thrown exception object e ; then, a series of checks determine if e has type that conforms to any of the handled exception types; if it does, execution jumps to the corresponding handler. As part of this transformation, BYTEBACK also prunes unreachable handlers: for every two exception types E_x, E_y such that $x < y$ and $E_y \leq E_x$ (E_y is a subtype of, or the same type as, E_x), it removes the check and handler for E_y since they are shadowed by the earlier check for E_x .

Transformation \mathcal{V}_{exc} also replaces the assignment $e := \text{@caught}$ that Jimple puts at the beginning of every handler with $e := \text{@thrown}; \text{@thrown} := \text{void}$, signaling that the current exception is handled, and thus the program will resume normal execution.

3.3.3 Exceptions in Method Calls. A called method may throw an exception, which the caller should propagate or handle. Accordingly, transformation \mathcal{V}_{exc} adds, after every method call instruction, code to check whether the caller set variable **@thrown** and, if it did, to handle the exception within the caller as if it had been directly thrown by it.

$$\mathcal{V}_{\text{exc}}(m(a_1, \dots, a_m)) = \left(\begin{array}{l} m(a_1, \dots, a_m); \\ \text{if } (\text{@thrown} = \text{void}) \text{ goto } \underline{\text{skip}}; \\ \mathcal{V}_{\text{exc}}(\text{throw } \text{@thrown}) \\ \underline{\text{skip}}: /* \text{code after call} */ \end{array} \right)$$

3.3.4 Potentially Excepting Instructions. Some bytecode instructions may implicitly throw exceptions when they cannot execute normally. In ??’s running example, $r.\text{read}()$ throws a `NullPointerException` if r is `null`; and the assignment to $a[i]$ throws an `IndexOutOfBoundsException` if i is not between 0 and $a.\text{length} - 1$. Transformation \mathcal{V}_{exc} recognizes such potentially excepting instructions and adds explicit checks that capture their implicit exceptional behavior. Let

op be an instruction that behaves normally if condition N_{op} holds, and throws an exception of type E_{op} otherwise; \mathcal{V}_{exc} transforms op as follows.

$$\mathcal{V}_{exc}(op) = \left(\begin{array}{l} \mathbf{if} \ N_{op} \ \mathbf{goto} \ \underline{normal}; \\ \mathcal{V}_{exc} \left(\begin{array}{l} e := \mathbf{new} \ E_{op}(); \\ \mathbf{throw} \ e; \end{array} \right) \\ \underline{normal}: \ op; \end{array} \right) \quad (1)$$

By chaining multiple checks, transformation \mathcal{V}_{exc} handles instructions that may throw multiple implicit exceptions. For example, here is how it encodes the potentially excepting semantics of array lookup $a[i]$, which fails if a is **null** or i is out of bounds.

$$\mathcal{V}_{exc}(res := a[i]) = \left(\begin{array}{l} \mathbf{if} \ (a \neq \mathbf{null}) \ \mathbf{goto} \ \underline{normal}_1; \\ \mathcal{V}_{exc} \left(\begin{array}{l} e_1 := \mathbf{new} \ \text{NullPointerException}(); \\ \mathbf{throw} \ e_1; \end{array} \right) \\ \underline{normal}_1: \ \mathbf{if} \ (0 \leq i \wedge i < a.length) \ \mathbf{goto} \ \underline{normal}_2; \\ \mathcal{V}_{exc} \left(\begin{array}{l} e_2 := \mathbf{new} \ \text{IndexOutOfBoundsException}(); \\ \mathbf{throw} \ e_2; \end{array} \right) \\ \underline{normal}_2: \ res := a[i]; \end{array} \right)$$

3.3.5 Normal and Exceptional Loop Exit. A loop may terminate normally (when the loop condition becomes false, or with a **break** statement) or exceptionally (when an exception thrown within the loop body propagates outside it). In some cases, verification requires distinguishing between a loop's exit points that correspond to normal or to exceptional termination. To support reasoning in these cases, transformation \mathcal{V}_{exc} adds an implicit loop invariant **@thrown = void** that is checked only at the *normal* exit points of a loop.

?? shows an example where this feature of BYTEBACK's translation is needed. The loop terminates normally when i reaches the value 7 (with a **break**) or 10 (with the loop condition becoming false); after termination, the whole enclosing method m also returns normally. In contrast, the loop terminates with an exception when i reaches the value 0; the exception is caught by the **try** block wrapping the loop, which then suppresses the exception so that method m still returns normally. Verifying the postcondition **@Return**—the method returns normally in all cases⁶—requires to check all possible return points: while it's clear that **@thrown = void** after the exception handling block executes, we also need to check that **@thrown = void** after the normal loop termination points. Hence, method m verifies correctly only after BYTEBACK has added the implicit loop invariant

⁶The loop does not terminate when $i > 10$ initially. BYTEBACK only checks partial correctness, and hence nonterminating behaviors are ignored.

<pre> @Return public void m(int i) { try { while (i ≠ 10) { i++; if (i = 0) throw new E(); if (i = 7) break; } } catch (E e) { assertion(eq(i, 0)); return; } assertion(eq(i, 10) eq(i, 7)); } </pre>	<pre> @Ensure(@thrown = void) public void m() { E e; @thrown := void; head: if (i = 10) goto exit; if (i ≠ 0) goto skip; @thrown = new E(); goto handler; skip: if (i ≠ 7) goto exit; goto head; exit: assert i = 10 ∨ i = 7; return; handler: e := @thrown; @thrown := void; assert i = 0; return; } </pre>	<pre> @Ensure(@thrown = void) public void m() { E e; @thrown := void; head: assume @thrown = void; if (i = 10) goto exit; if (i ≠ 0) goto skip; @thrown = new E(); goto handler; skip: if (i ≠ 7) goto exit; assert @thrown = void; goto head; exit: assert @thrown = void; assert i = 10 ∨ i = 7; return; handler: e := @thrown; @thrown := void; assert i = 0; return; } </pre>
---	---	---

Fig. 9. A Java loop that may terminate normally and exceptionally (left), and its Vimp encoding before (middle) and after (right) \mathcal{V}_{exc} introduces assertions that characterize normal loop exit points.

@thrown = void through all normal executions of the loops (?? right) whereas it fails verification without such additional information.

3.4 Order of Transformation

As outlined in ??, BYTEBACK applies the transformations \mathcal{V} from Jimple to Vimp in a precise order that incrementally encodes the full program semantics respecting dependencies.

Like the underlying bytecode that it represents, the Jimple code on which BYTEBACK operates is a form of three-address code, where each instruction performs one elementary operation (such as a call, a dereferencing, or a unary or binary arithmetic operation). BYTEBACK applies each transformation to incrementally translate Jimple into Vimp:

- i) BYTEBACK first applies \mathcal{V}_{exc} to make the exceptional control flow explicit.
- ii) Then, it applies \mathcal{V}_{ins} to every instruction, including, in particular, assertion instructions (??). In turn, \mathcal{V}_{ins} relies on transformations \mathcal{V}_{exp} to translate expressions within instructions, and \mathcal{V}_{agg} to aggregate specification expressions (??).

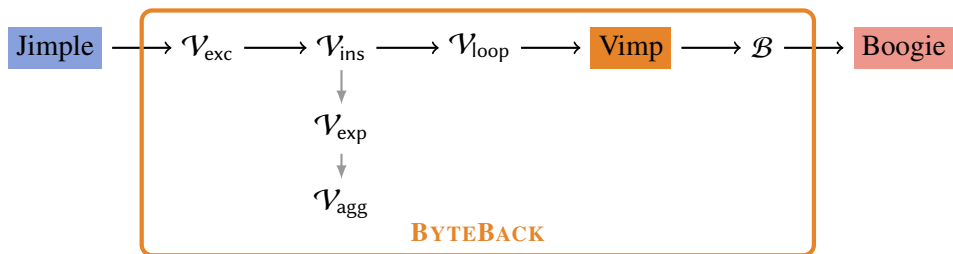


Fig. 10. BYTEBACK applies transformations in a definite order to incrementally translate Jimple to Vimp, and then Boogie.

- iii) Finally, it applies \mathcal{V}_{loop} to encode loop invariants as intermediate assertions; since this transformation is applied after \mathcal{V}_{exc} , the loop invariants can be checked at all loop exit points—normal and exceptional.

3.4.1 Boogie Encoding. The very last step \mathcal{B} of BYTEBACK’s pipeline takes the fully transformed Vimp program and encodes it as a Boogie program. Thanks to Vimp’s design, and to the transformation \mathcal{V} applied to Jimple, the Vimp-to-Boogie translation is mostly straightforward. A significant change pertains to the Boogie heap model:

- i) A global variable `@heap`: Heap stores the heap map: a mapping from references and fields to values.
- ii) Every `@Behavior` $C.p(a_1, \dots, a_m)$ becomes a Boogie (pure) function with an additional argument `heap`: **function** $C.p(heap: \text{Heap}, a_1, \dots, a_m)$, which is a reference to the heap. Correspondingly, any specification expression that refers to $C.p$ translates to a Boogie function application with global heap variable `@heap` as actual first argument.
- iii) Every `@TwoState @Behavior` $C.q(a_1, \dots, a_m)$ becomes a Boogie (pure) function with an additional argument `heap'`: **function** $C.q(heap: \text{Heap}, heap': \text{Heap}, a_1, \dots, a_m)$, which is a reference to the heap upon method entry. Correspondingly, any specification expression that refers to $C.q$ translates to a Boogie function application with `old(@heap)` as actual second argument.

As part of its encoding, BYTEBACK also generates a detailed Boogie axiomatization of all logic functions used to model the heap, as well as other parts of JVM execution—which we described in greater detail in previous work [?]. Another important addition is an axiomatization of subtype relations among exception types, used by Boogie’s `instanceof` function to mirror the semantics of the homonymous Java operator. Consider the whole tree T of exception types used in the program:⁷ each node is a type, and its children are its direct subtypes. For every node C in the

⁷Since the root of all exception types in Java is `class Throwable`—a concrete class—an exception type cannot be a subtype of multiple exception classes, and hence T is strictly a tree.

tree, BYTEBACK produces one axiom asserting that every child X of C is a subtype of C ($X \leq C$), and one axiom for every pair X, Y of C 's children asserting that any descendant types x of X and y of Y are *not* related by subtyping (in other words, the subtrees rooted in X and Y are disjoint): $\forall x, y: \text{Type} \bullet x \leq X \wedge y \leq Y \implies x \not\leq y \wedge y \not\leq x$.

3.5 Implementation Details

3.5.1 BLib as Specification Language. We are aware that BLib's syntax and conventions may be inconvenient at times; they were designed to deal with the fundamental constraints that any specification must be expressible in the source code and still be fully available for analysis in bytecode after compilation. This rules out the more practical approaches (e.g., comments) adopted by source-level verifiers. More user-friendly notations could be introduced on top of BLib—but doing so is outside the present paper's scope.

3.5.2 Class Invariants. BYTEBACK also offers basic support for class invariants. In BLib, a class C annotated with `@Invariant("inv")` declares that behavior method `inv` defines C 's class invariant. BYTEBACK simply adds `inv` as an assumption (a **free requires** in Boogie) at the beginning of every instance method of C , and as an additional postcondition of every instance method and constructor of C . This semantics for class invariants is admittedly quite primitive, but it is still sufficient to express fundamental validity conditions of objects of the same class C . In future work, we plan to equip BYTEBACK with substantially more refined invariant methodologies.

3.5.3 Attaching Annotations. As customary in deductive verification, BYTEBACK models calls using the modular semantics, whereby every called method needs a meaningful specification of its effects within the caller. To support more realistic programs that call to Java's standard library methods, BLib supports the `@Attach` annotation: a class S annotated with `@Attach(I.class)` declares that any specification of any method in S serves as specification of any method with the same signature in I . We used this mechanism to model the fundamental behavior of widely used methods in Java's, Scala's, and Kotlin's standard libraries. As a concrete example, we specified that the constructors of common exception classes do not themselves raise exceptions.

3.5.4 Implicit Exceptions. ?? describes how BYTEBACK models potentially excepting instructions. The mechanism is extensible, and the current implementation supports the five widespread kinds of implicit exceptions listed in ??:

- `NullPointerException` exceptions are thrown whenever dereferencing a `null` reference;
- `IndexOutOfBoundsException` exceptions are thrown whenever accessing an array with an index outside its bounds;
- `NegativeArraySizeException` exceptions are thrown when creating an array with a negative size;

EXCEPTING INSTRUCTION	EXCEPTION	CONDITION
dereferencing	$o \dots, o[i]$	<code>NullPointerException</code> $o \neq \text{null}$
array access	$a[i]$	<code>IndexOutOfBoundsException</code> $0 \leq i \wedge i < a.\text{length}$
array creation	<code>new t[n]</code>	<code>NegativeArraySizeException</code> $n \geq 0$
integer division	$v1 / v2$	<code>ArithmeticException</code> $v2 \neq 0$
cast	$(C) o$	<code>ClassCastException</code> $o \text{ instanceof } C \wedge o \neq \text{null}$

Table 4. Potentially excepting instructions currently supported by BYTEBACK.

- Arithmetic exceptions are thrown when dividing an integer by zero;
- ClassCast exceptions are thrown when a cast tries to convert between types that are not related by inheritance.

Users can choose among three ways of configuring these checks for implicitly thrown exceptions: *i*) the default is to model, and allow, potentially excepting instructions as shown in transformation (??); *ii*) alternatively, BYTEBACK can model potentially excepting instructions, as well as check that such exceptions never occur: in this case, an `assert false` replaces the block immediately following `goto normal` in transformation (??); *iii*) BYTEBACK can also just not model implicit exceptions: in this case, verification becomes unsound if the program may exhibit implicit exception behavior.

3.5.5 Dependency Analysis. BYTEBACK is implemented as a command-line tool that takes as input a classpath and a set E of class files within that path. The analysis collects the class signatures in all classes $D(E)$ on which the entry classes E transitively depend—where “ A depends on B ” means that A inherits from, or is a client of, B . After collecting such information about all classes in $E \cup D(E)$, BYTEBACK feeds them through its verification toolchain (??) that translates them to Boogie. BYTEBACK only verifies the *implementations* of the entry classes in E listed explicitly by the user; the implementations of $D(E)$ are ignored (i.e., not translated to Boogie for verification), but their interfaces and any specifications are still translated to reason about their usages in E .

3.5.6 Features and Limitations. The main limitations of BYTEBACK’s previous version [?] were a lack of support for exception handling and `invokedynamic`. As discussed in the rest of the paper, BYTEBACK now fully supports reasoning about exceptional behavior. We also added a, still limited, support for `invokedynamic`: any instance of `invokedynamic` is conservatively treated as a call whose effects are unspecified; furthermore, we introduced ad hoc support to reason about concatenation and comparison of string literal—which are implemented using `invokedynamic` since Java 9.⁸ A full support of `invokedynamic` still belongs to future work.

⁸<https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/StringConcatFactory.html>

Other remaining limitations of BYTEBACK’s current implementation are a limited support of string objects, and no modeling of numerical errors such as overflow (i.e., numeric types are encoded with infinite precision). Adding support for all of these features is possible by extending BYTEBACK’s current approach.

4 EXPERIMENTS

We demonstrated BYTEBACK’s capabilities by running its implementation on a collection of annotated programs involving exceptional behavior in Java, Scala, and Kotlin.

4.1 Programs

?? summarizes the key features of the 60 programs that we prepared to demonstrate BYTEBACK’s capabilities of reasoning about exceptional behavior; all these programs involve *some* exceptional behavior in different contexts.⁹ About half of the programs (31/60) are written in Java: 28 only use language features that have been available since Java 8, and another 3 rely on more recent features available since Java 17. To demonstrate how targeting bytecode makes BYTEBACK capable of verifying, at least in part, other JVM languages, we also included 15 programs written in Scala (version 2.13.8 of the language), and 14 programs written in Kotlin (version 1.8.0).

Each program consists of one or more classes with their dependencies, which we annotated with BBlib to specify exceptional and normal behavior, as well as other assertions needed for verification (such as loop invariants). The examples total 9 121 lines of code and annotations, with 386 annotations of different kinds, and 1 366 methods (including BBlib specification methods) involved in the verification process. According to their characteristics, the experiments can be classified into two groups: feature experiments and algorithmic experiments.

4.1.1 Feature Experiments. The programs listed in ?? are *feature* experiments: each of them exercises a small set of exception-related language features; correspondingly, their specifications check that BYTEBACK’s verification process correctly captures the source language’s semantics of those features. For example, experiments ??, ??, and ?? feature different combinations of try-catch blocks and throw statements that can be written in Java, Scala, and Kotlin, and test whether BYTEBACK correctly reconstructs all possible exceptional and normal execution paths that can arise. A more specialized example is experiment ??, which verifies the behavior of loops with both normal and exceptional exit points.

4.1.2 Algorithmic Experiments. The programs listed in ?? are *algorithmic* experiments: they implement classic algorithms in a way that also involves some exception behavior—for example,

⁹We focus on these exception-related programs, but the latest version of BYTEBACK also verifies correctly the 40 other programs we introduced in previous work to demonstrate its fundamental verification capabilities.

LANGUAGE		ENCODING	VERIFICATION	SOURCE	BOOGIE	METHODS	ANNOTATIONS		
		TIME [s]		SIZE [LOC]			<i>B</i>	<i>S</i>	<i>E</i>
<i>Java 8, 17</i>	total	14.1	29.2	7 274	193 563	1 071	149	51	181
	average	0.5	0.9	235	6 244	35	5	2	6
<i>Scala 2.13</i>	total	6.8	12.4	956	86 223	141	65	25	52
	average	0.5	0.8	64	5 748	9	4	2	3
<i>Kotlin 1.8</i>	total	6.3	12.2	891	82 117	154	64	27	50
	average	0.5	0.9	64	5 866	11	5	2	4
<i>all languages</i>	total	27.2	53.7	9 121	361 903	1 366	278	103	283
	average	0.5	0.9	152	6 032	23	5	2	5

Table 5. Summary of BYTEBACK’s verification experiments of exceptional behavior. For each LANGUAGE (Java, Scala, Kotlin, and all languages together), the table reports the **total** and **average** wall-clock time (in seconds) taken for ENCODING bytecode into Boogie, and for the VERIFICATION of the Boogie program; the size (in non-empty lines of code) of the SOURCE program with its annotations, and of the generated BOOGIE program; the number of METHODS that make up the program and its `BBLib` specification; and the number of ANNOTATIONS introduced for verification, split among: behavior methods *B* (`@Behavior`), pre- and postconditions *S* (`@Require`, `@Ensure`), and exception annotations *E* (`@Raise`, `@Return`).

using exceptions to signal when their inputs are invalid. The main difference between feature and algorithmic experiments is specification: algorithmic experiments usually have more complex pre- and postconditions than feature experiments, which they complement with specifications of exceptional behavior on the corner cases. For example, experiments `??`, `??`, and `??` compute the greatest common divisor iteratively, using the subtraction-based version of Euclid’s algorithm; their “functional” postconditions specify that the value returned by the methods is the same as that given by a pure, functional recursive definition of the same algorithm, whereas other parts of their specification say that they will throw an exception to signal when their inputs are invalid. Experiment `??` is an extension of `??`’s running example, where the algorithm is a simple stream-to-array copy implemented in a way that may give rise to various kinds of exceptional behavior.

Experiments `??` and `??` are the most complex programs in our experiments: they include a subset of the complete implementations of Java’s `ArrayList` and `LinkedList` standard library classes,¹⁰ part of which we annotated with basic postconditions and a specification of their exceptional behavior (as described in their official documentation). In particular, `ArrayList`’s exceptional specification focuses on possible failures of the class constructor (for example, when given a negative number as initial capacity); `LinkedList`’s specification focuses on possible failures of some of the read methods (for example, when trying to get elements from an empty list). Thanks

¹⁰ <https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html>

#	EXPERIMENT	LANG	ENCODING	VERIFICATION	SOURCE	BOOGIE	MET	ANNOTATIONS		
			TIME [s]		SIZE [LOC]			B	S	E
1	Implicit Division By Zero	J 8	0.4	0.7	62	5 704	10	3	0	7
2	Implicit Index Out of Bounds	J 8	0.5	0.8	84	5 848	16	5	2	8
3	Implicit Invalid Cast	J 8	0.4	0.7	63	5 692	36	3	0	8
4	Implicit Negative Array Size	J 8	0.4	0.7	50	5 668	8	3	0	4
5	Implicit Null Dereference	J 8	0.4	0.8	84	5 869	26	4	0	10
6	Check Division By Zero	J 8	0.4	0.7	62	5 704	10	3	0	7
7	Check Index Out of Bounds	J 8	0.5	0.8	84	5 848	16	5	2	8
8	Check Invalid Cast	J 8	0.4	0.7	63	5 692	36	3	0	8
9	Check Negative Array Size	J 8	0.4	0.7	50	5 668	8	3	0	4
10	Check Null Dereference	J 8	0.4	0.8	84	5 869	26	4	0	10
11	Multi-Catch	J 8	0.4	0.7	68	5 785	10	1	1	4
12	Throw-Catch	J 8	0.5	0.9	156	5 954	46	10	0	16
13	Throw-Catch in Loop	J 8	0.4	0.7	97	5 818	11	1	0	9
14	Try-Finally	J 8	0.5	0.8	125	5 797	15	2	4	6
15	Try-With-Resources	J 8	0.5	0.8	197	6 733	26	3	4	13
16	Try-With-Resources on Local	J 17	0.4	0.6	43	5 635	6	1	1	1
17	Implicit Division By Zero	S	0.4	0.7	62	5 719	10	3	0	7
18	Implicit Index Out of Bounds	S	0.4	0.7	44	5 698	7	2	1	4
19	Implicit Invalid Cast	S	0.4	0.7	42	5 680	7	2	0	4
20	Implicit Negative Array Size	S	0.4	0.7	51	5 678	8	3	0	5
21	Implicit Null Dereference	S	0.4	0.6	43	5 677	6	1	0	6
22	Multi-Catch	S	0.5	0.6	46	5 748	7	1	1	2
23	Throw-Catch	S	0.5	0.8	116	5 903	22	8	0	13
24	Try-Finally	S	0.5	0.8	117	5 847	15	2	4	3
25	Implicit Division By Zero	K	0.4	0.7	62	5 719	10	3	0	7
26	Implicit Index Out of Bounds	K	0.4	0.7	44	5 698	7	2	1	4
27	Implicit Invalid Cast	K	0.4	0.7	42	5 680	7	2	0	4
28	Implicit Negative Array Size	K	0.4	0.7	51	5 678	8	3	0	5
29	Implicit Null Dereference	K	0.4	0.6	43	5 677	6	1	0	6
30	Throw-Catch	K	0.5	0.9	110	5 924	22	8	0	12
31	Try-Finally	K	0.5	0.8	108	5 843	15	2	4	3
total			13.9	22.9	2 357	179 770	484	97	28	202
average			0.4	0.7	76	5 799	16	3	1	7

Table 6. Verification experiments that demonstrate BYTEBACK’s support for exception-related features. Each row corresponds to a *feature experiment* that verified one program with exceptional behavior. The columns are as in ?? (the LANGUAGES are abbreviated as J 8 for Java 8, J 17 for Java 17, S for Scala 2.13, K for Kotlin 1.8).

to BBLib’s features (including the `@Attach` mechanism described in ??), we could add annotations without modifying the implementation of these classes. Note, however, that we verified relatively simple specifications, focusing on exceptional behavior; a dedicated support for complex data structure functional specifications [??] exceeds BBLib’s current capabilities and belongs to future work.

#	EXPERIMENT	LANG	ENCODING	VERIFICATION	SOURCE	BOOGIE	MET	ANNOTATIONS		
			TIME [s]		SIZE [LOC]			B	S	E
32	Binary Search	J 8	0.4	0.7	52	5 680	6	4	4	1
33	Counter	J 8	0.4	0.7	51	5 729	6	1	1	4
34	GCD	J 8	0.4	0.6	48	5 697	6	4	1	1
35	Linear Search	J 8	0.4	0.6	62	5 657	9	6	6	2
36	Max (double)	J 8	0.4	0.8	46	5 729	6	4	3	1
37	Max (int)	J 8	0.4	0.8	46	5 725	6	4	3	1
38	Selection Sort (double)	J 8	0.5	1.9	97	5 882	14	12	3	1
39	Selection Sort (int)	J 8	0.4	1.7	97	5 877	14	12	3	1
40	Sequences	J 8	0.4	0.7	38	5 711	4	1	1	2
41	Square of Sorted Array	J 8	0.4	0.6	61	5 691	7	5	1	1
42	Sum	J 8	0.4	0.6	44	5 678	5	3	1	1
43	ArrayList	J 8	0.6	3.1	2 630	13 974	288	14	0	24
44	LinkedList	J 8	0.6	3.0	2 479	11 814	371	9	4	17
45	Summary	J 17	0.4	0.6	48	5 661	5	3	2	1
46	Read Resource	J 17	0.4	0.8	102	6 091	18	13	4	6
47	Counter	S	0.5	0.7	49	5 703	8	4	3	3
48	GCD	S	0.5	0.6	45	5 708	5	3	1	1
49	Linear Search	S	0.5	0.6	46	5 609	6	4	3	1
50	Max (double)	S	0.5	0.8	47	5 738	6	4	3	0
51	Max (int)	S	0.5	0.7	48	5 735	6	4	3	1
52	Selection Sort (double)	S	0.5	1.9	100	5 892	14	12	3	1
53	Selection Sort (int)	S	0.5	1.5	100	5 888	14	12	3	1
54	Counter	K	0.4	0.7	47	5 720	8	4	3	3
55	GCD	K	0.4	0.6	44	5 728	5	3	1	1
56	Linear Search	K	0.4	0.6	43	5 795	6	4	3	1
57	Max (double)	K	0.5	0.7	48	5 924	6	4	3	1
58	Max (int)	K	0.4	0.7	48	5 947	5	4	3	1
59	Selection Sort (double)	K	0.5	2.0	99	6 073	14	12	3	1
60	Selection Sort (int)	K	0.5	2.0	99	6 073	14	12	3	1
total		13.3	30.9	6 764	182 133	882	181	75	81	
average		0.5	1.1	233	6 280	30	6	3	3	

Table 7. Verification experiments that demonstrate BYTEBACK’s support for verifying algorithms involving exceptional behavior. Each row corresponds to an *algorithmic experiment* that verified one program with exceptional behavior. The columns are as in ?? (the LANGUAGES are abbreviated as J 8 for Java 8, J 17 for Java 17, S for Scala 2.13, K for Kotlin 1.8).

4.1.3 Implicit Exceptions. As explained in ??, users of BYTEBACK can enable or disable checking of implicitly thrown exceptions. Java experiments ??–??. Scala experiments ??–??. and Kotlin experiments ??–?? demonstrate verifying exceptional behavior triggered by division by zero (ArithmeticException), out-of-bounds array access (IndexOutOfBoundsException), conversion to incompatible types (ClassCastException), and the ubiquitous **null**-pointer dereferencing

(`NullPointerException`). Java experiments ??–?? verify instead the *absence* of implicit exceptional behavior—instead of accepting it as an acceptable method behavior as in the previously mentioned experiments. Similarly, Java experiment ?? involves implicit exception behavior, whereas Java experiments ??–??. Scala experiments ??–??. and Kotlin experiments ??–?? verify that no implicit exception occurs.

4.2 Results

All experiments ran on a Fedora 36 GNU/Linux machine with an Intel Core i9-12950HX CPU (4.9GHz), running Boogie 2.15.8.0, Z3 4.11.2.0, and Soot 4.3.0. To account for measurement noise, we repeated the execution of each experiment five times and report the average wall-clock running time of each experiment, split into `BYTEBACK` bytecode-to-Boogie encoding and Boogie verification of the generated Boogie program. We ran Boogie with default options except for experiment ??, which uses the `/infer:j` option (needed to derive the loop invariant of the enhanced `for` loop, whose index variable is implicit in the source code).

All of the experiments verified successfully. To sanity-check that the axiomatization or any other parts of the encoding introduced by `BYTEBACK` are consistent, we also ran Boogie’s so-called smoke test on the experiments;¹¹ these tests inject `assert false` in reachable parts of a Boogie program, and check that none of them pass verification.

As you can see in ??, `BYTEBACK`’s running time is usually below 1.5 seconds; and so is Boogie’s verification time. Unsurprisingly, programs ?? and ?? are outliers, since they are made of larger classes with many dependencies; these slow down both `BYTEBACK`’s encoding process and Boogie’s verification, which have to deal with many annotations and procedures to analyze and verify.

As part of the extensions and improvements we introduced for this paper, the latest version of `BYTEBACK` is more seamlessly integrated within the Soot framework, and hence performs significantly better: the running time of `BYTEBACK` (column `ENCODING` in ??) is more than halved (down to 0.5 seconds per program on average from [?]’s 1.3 seconds), and also the Boogie running time (column `VERIFICATION` in ??) has improved (down to 0.9 seconds per program on average from [?]’s 1.2 seconds) as a result of a streamlined encoding.

5 RELATED WORK

The state-of-the-art deductive verifiers for Java include OpenJML [?], KeY [?], and Krakatoa [?]; they all process the source language directly, and use variants of JML specification language—which offers support for specifying exceptional behavior.

¹¹Smoke tests provide no absolute guarantee of consistency, but are often practically effective.

Exceptional Behavior Specifications. Unlike BBLib, where postconditions can refer to both exceptional and normal behavior, JML clearly separates between the two, using **ensures** and **signals** clauses (as demonstrated in ??). These JML features are supported by OpenJML, KeY, and Krakatoa according to their intended semantics.

```

//@ ensures this.a == a;
//@ signals (Throwable) this.a == a;
public void m(int a)
{ this.a = a;
  if (ε) throw new RuntimeException(); }

@Ensure(this.a = a)
public void m(int a)
{ this.a = a;
  if (ε) throw new RuntimeException(); }

```

Fig. 11. Equivalent exceptional specifications in JML (left) and BBLib (right).

Implicit Exceptional Behavior. Implicitly thrown exceptions, such as those occurring when accessing an array with an out-of-bounds index, may be handled in different ways by a verifier: *i)* ignore such exceptions; *ii)* implicitly check that such exceptions never occur; *iii)* allow users to specify these exceptions like explicit ones. OpenJML and Krakatoa [?] follow strategy ??, which is sound but loses some precision since it won't verify some programs (such as ??'s example); KeY offers options to select any of these strategies, which gives the most flexibility; Similarly BYTEBACK also offers the ability to select any of these strategies, so that users can decide how thorough the analysis of exceptional behavior should be.

Java Exception Features. OpenJML, KeY, and Krakatoa [?] all support try-catch-finally blocks, which have been part of Java since its very first version. The first significant extension to exceptional feature occurred with Java 7, which introduced multi-catch and try-with-resources blocks.¹² KeY and Krakatoa support earlier versions of Java, and hence they cannot handle either feature. OpenJML supports many features of Java up to version 8, and hence can verify programs using multi-catch or try-with-resources—with the exception of try-with-resources using an existing **final** variable, a feature introduced only in Java 9. As usual, our point here is not to criticize these state-of-the-art verification tools, but to point out how handling the proliferation of Java language features becomes considerably easier when targeting bytecode following BYTEBACK's approach.

Other Deductive Verifiers. The few other deductive verifiers for Java typically focus on features of the Java language other than exceptions [?]. VerCors [?] is a verifier for concurrent and distributed software using separation logic for specification; it supports pre-Java 7 exception features¹³ such

¹² <https://www.oracle.com/java/technologies/javase/jdk7-relnotes.html>

¹³ <https://github.com/utwente-fmt/vercors/wiki/Exceptions-&-Goto#support>

as **try/catch/finally** blocks, and some JML features to specify exceptional behavior (mainly, the **signals** clause), with some restrictions¹⁴ such as that any exception that can be thrown must be mentioned in a **throws** or **signals** clause. VeriFast [??] is a modular verifier for C and Java programs annotated with separation logic specifications; VeriFast supports “most of Java 1.0”¹⁵—in particular, **try/catch/finally** blocks and **throws** clauses. Plural [??] is a modular static checker of object protocols specified using tpestates and fractional permissions; given this focus, it lacks support for some Java features such as exceptions [?, § 2.3].

Intermediate Representation Verifiers. A different class of verifiers—including JayHorn [??], SeaHorn [?], and SMACK [?]—target intermediate representations (JVM bytecode for JayHorn, and LLVM bitcode for SeaHorn and SMACK). Besides this similarity, these tools’ capabilities are quite different from BYTEBACK’s: they implement analyses based on model-checking (with verification conditions expressible as constrained Horn clauses, or other specialized logics), which provide a high degree of automation (e.g., they do not require loop invariants) to verify simpler, lower-level properties (e.g., reachability). Implicitly thrown exceptions are within the purview of tools like JayHorn, which injects checks before each instruction that may dereference a null pointer, access an index out of bounds, or perform an invalid cast. In terms of usage, this is more similar to a specialized static analysis tool that checks the absence of certain runtime errors [??] than to fully flexible, but onerous to use, deductive verifiers like BYTEBACK.

BML [?] is a specification language for bytecode; since it is based on JML, it is primarily used as a way of expressing a high-level Java behavioral specification at the bytecode level. This is useful for approaches to proof-carrying code [?] and proof transformations [?], where one verifies a program’s source-code and then certifies its bytecode compilation by directly transforming the proof steps.

6 CONCLUSIONS AND FUTURE WORK

This paper presented the latest extension of the BYTEBACK verification approach, which equipped it with support to reasoning about exceptional behavior of Java, as well as of other JVM languages like Scala and Kotlin.

6.1 Bytecode Level vs. Source Level

Reasoning about exceptional behavior (as well as about other behavioral features) at the level of Java bytecode can complement the more traditional (and mature) approaches that work at the level of source code.

¹⁴<https://vercors.ewi.utwente.nl/wiki/#exceptions>

¹⁵https://verifast.github.io/verifast-docs/java_compilation_units.html

The main advantages of BYTEBACK’s approach—working at the bytecode level—are its flexibility and robustness with respect to changes and additions of source-level features. An intermediate representation like bytecode is invariably more stable (in terms of features and their semantics) than a feature-laden, modern programming language like Java. As a result, BYTEBACK can seamlessly analyze programs written in any version of Java (and even in other JVM languages), whereas source-level verifiers such as KeY and OpenJML are limited to analyzing earlier versions of Java. Even when updating a source-level verifier to handle a new language feature would be conceptually straightforward, every new language release aggravates the burden on the verifier’s developers, and diverts effort that could be better spent on improving the verification capabilities on the currently supported Java feature set.

At the same time, working at the level of an intermediate representation has its own challenges, which stem from the wide abstraction gap between the source level and the bytecode level. Whereas a source-level verifier can use comments, or any other convenient, high-level notation to express specifications, bytecode-level verification is limited to formats that are expressible as code and “survive” compilation to bytecode. BYTEBACK introduced BBlib to this effect; while BBlib annotations do the job, they tend to be more cumbersome and verbose than if one used a custom specification language such as JML. Another disadvantage of targeting bytecode-level verification is that it may make debugging failed verification attempts harder: if Boogie fails when verifying a program produced by BYTEBACK, the location and nature of the failure may be hard to trace back to the source level. In particular, there are features of the generated Boogie programs that are just a figment of BYTEBACK’s encoding (for example, the explicit exceptional control flow, which is implicit at the source level) or of the compilation to bytecode (for example, a **for**-each loop in Java gets compiled down to a bytecode loop with an explicit loop counter variable, which doesn’t exist in the source code).

Overall, BYTEBACK’s approach is meant to complement—rather than replace—the core work in source-level deductive verification, and make it readily available to the latest languages and features.

6.2 Future Work

In future work, we plan to address several of BYTEBACK’s main limitations:

- BYTEBACK currently only offers a very primitive support for class invariants, which severely limits the kinds of properties of complex object structures that it can verify. In future work, we would like to equip BYTEBACK with a modern invariant methodology [?????????].
- While BYTEBACK’s current version has some capability of reasoning about invokedynamic calls, it remains too limited to verify advanced functional features. Here, part of the challenge

is devising a suitable notation that can be used to specify the behavior of function objects in a genuinely modular way.

- A natural direction to make `BYTEBACK` more palatable to the end user is providing a specification notation that is more natural and concise than `BLib`. As discussed in the paper, we developed `BLib` with the primary goal of being fully reusable with any JVM-compatible language, and fully available at the bytecode level without requiring any processing of the source code on top of what is done by the standard compilers. We are definitely open to the idea of relaxing some of these constraints to add support for more standard notations (JML is the prime example [?]), provided `BLib` remains available as a lingua franca for all situations where its greater flexibility is needed.
- While `Boogie` remains widely used as an intermediate verification language, it could be interesting to explore alternatives—in particular, the well-known `Why3` platform [?]. `Why3` is based on the `WhyML` language, a member of the ML family of functional languages. Some of its features (such as exception handling, algebraic data types, termination checks, and a modular library of theories) make it a somewhat complementary alternative to `Boogie`, which could bring advantages to encode and verify features of bytecode such as exceptions, and the aforementioned functional features.

ACKNOWLEDGMENTS

Work partially supported by SNF grant 200021-207919 (LastMile).

Temporary page!

\LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because \LaTeX now knows how many pages to expect for this document.