Reasoning about Substitutability at the Level of JVM Bytecode*

Marco Paganoni[1] and Carlo A. Furia[1 [0000−0003−1040−3201]]

Software Institute, USI Università della Svizzera italiana, Lugano, Switzerland
marco.paganoni@usi.ch bugcounting.net

Abstract. Subtyping in object-oriented languages is widely based on Liskov's substitution principle, which offers static correctness guarantees of type safety while abstracting implementation details. Unfortunately, the type systems of languages like Java cannot statically enforce full behavioral substitutability, and in fact there are numerous examples of libraries some of whose components are related by inheritance but not substitutable (for example, because they do not implement "optional" operations).
In this paper, we present a novel approach to precisely specify and reason about substitutability in JVM languages. A distinctive feature of our approach is that it targets JVM bytecode, as opposed to a program's source code, as it is based on the BYTEBACK deductive verifier. To support reasoning about substitutability, we extended BYTEBACK with ghost specifications, a (restricted) form of class invariants, and substitutability-preserving specification inheritance (precondition weakening and postcondition strengthening). Equipped with these features, BYTEBACK can now reason precisely about behavioral substitutability violations in a way that is applicable to realistic examples (such as with optional operations of Java's `List` interface). Our experiments also demonstrate that BYTEBACK can analyze substitutability in programs written in a combination of JVM languages, including multi-language code where Scala or Kotlin code interacts with Java libraries.

1 Introduction

Since Liskov and Wing's influential work [17], behavioral *substitutability* is a widely accepted, fundamental requirement of types related by subtyping—especially relevant for object-oriented programming [20]. Informally, a type T is substitutable for another type S if replacing an instance of S with an instance of T does not break the behavior of the code using it. Thus, a type system that can statically check substitutability combines correctness guarantees (absence of behavioral incompatibility errors) with flexibility (transparently switching between different implementations through polymorphism). This is why most modern statically typed object-oriented languages are designed with type systems that enforce substitutability.

Unfortunately, other features of a programming language may still allow developers to bypass a type system's static checks and introduce subtypes that are not truly substitutable. Take the example of Java: the overriding of a method `m` in a class `D` that inherits

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

from another class `C` can simply throw an `UnsupportedOperation` exception; then, `D` is not substitutable for `C`, because the call `o.m()` terminates normally if `o` is an instance of `C`, and exceptionally if it is an instance of `D`—even though `D` is a subtype of `C` because they are related by inheritance. This substitutability-breaking pattern is not merely a theoretical possibility: there is empirical evidence that it is (deliberately) adopted in popular Java libraries to selectively enable or disable inherited operations [27,18,19].

In this paper, we introduce an approach to model behavioral substitutability constraints within a deductive verification framework, which we use to precisely reason about the behavior of programs involving these features. We develop our approach on top of our BYTEBACK verifier [23,22], which works on JVM bytecode. Extending BYTEBACK makes our approach applicable to different languages that run on the JVM, such as (any versions) of Java, as well as (subsets of) Scala and Kotlin. This capability is especially relevant when verifying Scala or Kotlin programs that *use* Java libraries with optional operations, such as in Sec. 2's capsule example; using our approach, we are able to consistently check substitutability violations (or to prove their absence) even in multi-language programs.

To support reasoning about substitutability, we extend BYTEBACK with three specification features, which we present in detail in Sec. 3: *i*) Ghost specification predicates, which we use as "flags" denoting whether a certain operation is or is not available; *ii*) A (restricted) form of class invariants, which specify, in each concrete class, which specification predicates hold; *iii*) Substitutability-preserving specification inheritance (precondition weakening and postcondition strengthening), which propagates the information about available operations through the inheritance hierarchy.

While these specification features are fairly standard in source-level deductive verifiers for object-oriented languages (for example, JML-based verifiers such as KeY [1] and OpenJML [6] support them), they become considerably more challenging to implement at the level of bytecode—in a way that is applicable to multi-language programs, and without access to any source code of clients or libraries. As we discuss in Sec. 3.6, our approach relies on mechanisms to "attach" a specification to existing classes in the system, as well as to correctly propagate such attached specifications to other classes that are related by inheritance.

The extension of BYTEBACK described in the present paper is, to our knowledge, the first deductive verification technique that can reason about behavioral properties of programs combining different JVM languages. Even when applied to single-language programs, working at the level of bytecode has the advantage of handling robustly any versions of Java (and other JVM languages); in contrast, as we discuss in Sec. 5, source-level deductive verifiers usually only support older versions of Java. Another distinguishing capability of our approach is verifying programs that use complex libraries (such as the JDK) by directly analyzing the compiled bytecode, without need to build or access the libraries' source code. Sec. 4 discusses several experiments that demonstrate these capabilities on benchmark examples in Java, Kotlin, and Scala.

Contributions and positioning. In summary, the paper makes the following contributions: *i*) Specification features to express substitutability properties on top of the type system of JVM languages; *ii*) A verification technique based on these specification features; *iii*) An implementation of the verification technique built on top of the BYTEBACK verifier [23]; *iv*) An experimental evaluation with 22 programs involving unsupported

operations in Java, Kotlin, and Scala; *v*) For reproducibility, our version of ByteBack and all experimental artifacts are available in a replication package [24].

While our implementation is based on ByteBack's previous instances [23,22], this paper's contributions substantially extend our previous work with specification and verification features necessary to reason about behavioral substitutability, and with a newly engineered implementation that makes them applicable to single- and multi-language bytecode programs. For simplicity, "ByteBack" will refer to the new verification technique, and its implementation, described in the rest of this paper—unless we explicitly point out that we are referring to earlier work.

```
List<Integer> ml = new ArrayList<Integer>();
// valid call: ml is mutable
ml.add(42);

List<Integer> il = List.of(1, 2);
// invalid call: il is immutable
il.add(42);
```

(a) The second call to method `add` is accepted by the Java compiler, but results in an exception at runtime because `List.of` returns an immutable `List`.

```
val bi: Buffer[Int] = java.util.List.of(1, 2).asScala
// invalid call: bi is immutable
bi.append(42)
```

(b) The call to method `append` is accepted by the Scala compiler, but results in an exception at runtime because Java's `List.of` returns an immutable `List`, but `asScala` converts all instances of `List` to instances of mutable `Buffer`.

```
@Attach(java.util.List)
interface ListSpec<T> {

 @NoState @Behavior
 boolean is_mutable();

 @Raise(UnsupportedOperationException.class, when = ¬is_mutable())
 @Return(when = is_mutable())
 boolean add(T element);

 @Abstract
 @Ensure( ¬result.is_mutable())
 static <E> List<E> of(E e1, E e2)
 { return null; }

}
```

(c) An excerpt of ByteBack's specification of `java.util.List`.

```
@Attach(java.util.ArrayList)
@Invariant(Ghost.of(
  ListSpec.class, this
 ).is_mutable())
public class ArrayListSpec<T> {
  // ...
}

@Attach(java.util.LinkedList)
@Invariant(Ghost.of(
  ListSpec.class, this
 ).is_mutable())
public class LinkedListSpec<T> {
  // ...
}
```

(d) Class invariants of `java.util`'s `ArrayList` and `LinkedList`.

```
@Require(Ghost.of(classOf[ListSpec[A]], this).is_mutable())
@Return
def asScalaBufferConverter[A](l: java.util.List[A]): AsScala[Buffer[A]]
```

(e) An excerpt of ByteBack's specification of Scala's `asScalaBufferConverter`, called by `asScala` to convert a Java `List` to a mutable Scala `Buffer`.

Fig. 1: An example of behavioral substitutability errors that are uncaught by the Java and Scala type systems, which can be precisely analyzed with this paper's deductive verification technique. The annotations use a simplified `BBlib` syntax, similar to the one used in previous work on ByteBack [23,22].

2 Motivating Example

The best-known example of Java libraries that violate behavioral substitutability is probably that of `java.util`'s collections—in particular, interface `List`. Operations such as

`add` are denoted *optional*, and hence some concrete classes inheriting from `List` may choose not to implement those operations, or to implement them only for a restricted set of valid inputs.

Fig. 1a shows a snippet of Java code that incurs this issue. First, there is a call of `add` on target `ml`, which is an instance of `ArrayList`. Since `ArrayList` is a subtype of `List` that implements all optional operations, the call `ml.add(42)` returns without errors. Then, there is a call of `add` on target `il`, which is an instance of `List` returned by static method `List.of`. As explained by the JDK's documentation, `List.of` always returns *immutable* lists;[a] therefore, the call `il.add(42)` fails with an `UnsupportedOperation` exception. However, the compiler accepts this code because, according to the type system's rules, every instance of `List` should support method `add`.

Fig. 1b demonstrates the same kind of problem in Scala code that uses Java data structures. Scala's library function `asScala` converts instances of `java.util.List` to instances of `mutable.Buffer`; this is usually sound, because Java lists are generally mutable. However, `List.of` actually returns an immutable list, which `asScala` simply wraps around; hence, calling modification operations, such as `append`, on instance `bi` of `Buffer` results in an exception at runtime. Even though Scala's own collection library has been carefully designed to avoid such behavioral substitutability problems, Scala code may still indirectly suffer from the limitations of Java's collections design.

To round off this brief diagnosis of the problem, we observe that the issue with using methods such as `List.of` is twofold: *i*) first, this method returns instances of `List` where some operations are not available (breaking `List`'s behavioral interface); *ii*) second, there is no type[1] corresponding to an immutable variant of `List` that we can explicitly convert `List.of`'s output to.

Specifying substitutability. Fig. 1 outlines how our approach supports precisely specifying different behavioral variants of `List`. Fig. 1c shows how we equip type `List` with a behavioral specification predicate `is_mutable()`, which returns **`true`** iff the current instance of `List` is indeed modifiable. Since we do not have access to the source code of `List`, ByteBack provides the **`@Attach`** annotation to augment any compiled class in the system with specification elements. Method `is_mutable()` is marked with ByteBack annotations **`@Behavior`** (it is used for behavioral specifications, and hence any of its implementations must be side-effect free) and **`@NoState`** (its value does not depend on the current instance of `List`'s state, since a list cannot become immutable after it is initialized).

After provisioning predicate `is_mutable()`, we can use it to specify any other members of `List`: Fig. 1c shows the specification of methods `add` and `of`.[2] For `add`, the specification says that it returns normally when `is_mutable()` is true, and with an exception when `is_mutable()` is false. For `of`, the specification says that it returns an instance **`result`** of `List` for which `is_mutable()` does not hold—which is how we specify that `List.of` returns unmodifiable lists. Since Java does not allow **`static abstract`** methods, we provide a dummy implementation of **`static`** method `List.of`, but the **`@Abstract`** annotation instructs ByteBack to effectively ignore it.

[1] While there is a class `ImmutableCollections` in the JDK,[b] it is not part of the public API, and hence it is inaccessible to clients.

[2] For simplicity, we only show the two-argument variant of `List.of`.

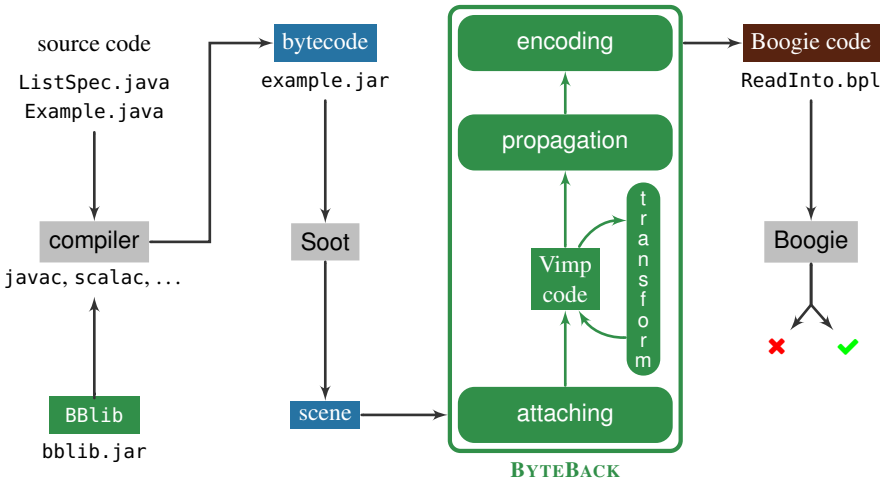

Fig. 2: An overview of BYTEBACK's verification workflow.

Fig. 1d lists other specifications that use `is_mutable()`. Precisely, we equip concrete classes `ArrayList` and `LinkedList`—both subclasses of `List`—with a class `@Invariant`: `is_mutable()` always holds for their instances. The syntax of these class invariants is somewhat cumbersome, as it uses a utility method `Ghost.of` that we designed as part of this paper's contributions. Since specification predicates such as `is_mutable()` are only supplied during BYTEBACK's analysis—after the Java program under verification has been compiled into bytecode—we cannot rely on Java's standard inheritance mechanisms to propagate specification elements to subclasses. Instead, BYTEBACK deploys a custom specification propagation algorithm, which weaves usages of `Ghost.of` into the type hierarchy during analysis. In Fig. 1d, `Ghost.of` is used to equip **`this`** (the current instance of `ArrayList` or `LinkedList`) with predicate `is_mutable()` declared in `ListSpec`.

Fig. 1e uses `is_mutable()` to specify a precondition for `asScalaBufferConverter`, to which `asScala` delegates the conversion of a Java `List` to a Scala mutable `Buffer`. Using in Scala the same mechanisms seen in `ArrayList`'s specification, Fig. 1e states that only mutable Java lists can be converted to mutable Scala buffers.

Verifying substitutability. Equipped with Fig. 1c and Fig. 1d's specifications, BYTEBACK can reason precisely about Fig. 1a's example. Thanks to `ArrayList`'s class invariant, it knows that `ml.add(42)` terminates normally; thanks to `List.of`'s postcondition, it knows that `il.add(42)` terminates by throwing an `UnsupportedOperation` exception—which may or may not be correct behavior, depending of whether the surrounding code expects that exception. BYTEBACK also issues a verification error for Fig. 1b's example: the call to `asScala` violates Fig. 1e's precondition, and hence it is invalid.

3 How BYTEBACK Specifies and Verifies Substitutability

Fig. 2 overviews how the deductive verification technique described in this paper works. Since our contributions is built atop BYTEBACK, the overall workflow remains similar

to the one introduced in ByteBack's previous work [23,22]—with some important differences that we outline in Sec. 3.1 and Sec. 3.5 and describe in greater detail in the other subsections.

3.1 ByteBack Specification: Overview

Users of ByteBack add specifications the source code by means of `BBlib`—ByteBack's source-level annotation library. Sec. 3.2 summarizes the core annotations provided by `BBlib` (such as pre- and postconditions), as well as those that are especially geared towards the present paper (such as class invariants). A key novel feature are *attached* specifications: users can list specification elements in separate *ghost* classes, and instruct ByteBack to weave them into the project under verification's implementation. Sec. 3.3 describes this mechanism, which is especially useful to supply specifications for components whose source code is not available (including system libraries). In an object-oriented language, subtyping is connected to class inheritance; correspondingly, Sec. 3.4 describes how `BBlib`'s specifications are also *inherited*, following rules that are consistent with behavioral subtyping (i.e., precondition weakening and postcondition strengthening).

3.2 `BBlib` Specification Elements

`BBlib` annotations include all fundamental behavioral specification elements; since they were introduced in the previous work on ByteBack, we only summarize them here to make the paper self contained. *Method* specification elements include preconditions (`@Require`) and postconditions (`@Ensure`), as well as annotations to specify exceptional vs. normal behavior: `@Raise(`E`, when =` c`)` specifies a method that throws an exception of class E when it is called on an object where condition c holds; conversely, `@Return(when =` c`)` specifies a method that returns normally when it is called on an object where condition c holds. For example, Fig. 1c specifies that method `List.add` terminates normally if **`this`**`.is_mutable()` holds, and with an `UnsupportedOperation` exception otherwise.

Method *bodies* may include generic assertions (`assertion(`c`)` leads to a verification error iff c does not hold when execution reaches it), assumptions (`assumption(`c`)` ignores all executions where c does not hold), and loop invariants (`invariant(`c`)` captures the fundamental inductive property of the loop where it appears).

Class-level specifications use annotation `@Invariant(`c`)`, which asserts that property c must hold for all instances of the specified class. For instance, Fig. 1d annotates class `ArrayList` with the invariant ℓ`.is_mutable()`, which every object ℓ of class `ArrayList` must satisfy. Such support for *class invariants* has been introduced in ByteBack only recently, as it is necessary to reason about substitutability. Nevertheless, ByteBack's class invariants are still limited in expressiveness: a class C with `@Invariant(`I`)` is simply equivalent to annotating every method of C with a "free"[3] precondition I and a (regular) postcondition I, and every constructor of C with a postcondition I. This simple class invariant semantics is sufficient for this paper's purposes, but it falls short of a full-fledged invariant *methodology* [15,16,28,5,12,26], which should support temporary violations of the class invariant (for example, by pri-

[3] A *free* precondition is an assumption about a method's pre-state.

vate methods), as well as expressing invariants that depend on the state of multiple objects.

`BBlib` expresses specification *predicates* as methods marked with `@Behavior`. BYTEBACK checks that every method `m` equipped with such annotation can be expressed as a purely logic predicate: *i*) `m` must return a **`boolean`**, *ii*) have a signature consistent with `m`, *iii*) be pure and aggregable,[4] and *iv*) only call other `@Behavior` methods. Annotation `@Behavior` generalizes the `@Predicate` annotation used in previous work [23,22], since `@Behavior` also allows recursive calls in a behavior method body. Other annotations further constrain what a behavior method can predicate over. By default, a behavior method `b` used to specify a method `m` has access to `m`'s arguments (possibly including its returned value, if `m` does not return **`void`**), as well as to the current object's state (e.g., through its fields). If `b` is annotated with `@NoState`, it cannot depend on the object state directly. Conversely, if `b` is annotated with `@TwoState`, it can also access the current object's *pre-state* (the state just before executing `m`). Thus, `@TwoState` behaviors are used in postconditions, where they relate the post-state to the pre-state. On the other hand, `@NoState` behaviors are useful to specify properties that hold "absolutely", such as `is_mutable()` in Fig. 1c: an instance of `List` is either mutable or immutable, and cannot change this property without being explicitly converted into a new object. Thus, annotation `@NoState` is especially useful for behavioral substitutability properties like those we reason about in this paper; declaring them as `@NoState` simplifies framing, since BYTEBACK only needs to check that their definitions do not refer to the object state.

3.3 Attached and Ghost Specifications

In previous work, all `BBlib` annotations had to be introduced in the source code of the element they refer to; for example, a method `m`'s postcondition had to appear just before `m`'s declaration in the program. This mechanism is limiting in all cases where we do not have access to a program's source code, or we simply do not want to alter it in any way. Fig. 1's example is a representative instance of this scenario: even if we had access to the source code of the JDK's collections library, recompiling them just to introduce a handful of specification elements would be practically very inconvenient. In this work, we introduce the *attaching* mechanism, which enables users to add specifications to any element of the program under verification, regardless of whether they have access to its source code.

Attached specifications. A class S annotated with `@Attach`(C) instructs BYTEBACK to apply all specification features introduced in S to class C. Precisely, the specification of any method `m` in S whose signature matches that of a method with the same name in C is used as specification of `m` in C. The attaching mechanism is compositional: users can attach multiple specification classes to the same class C—each specifying only a part of C's interface. If C includes some source-level specification of its own, users can selectively keep some of them, while overriding (or adding) the specifications of other methods. Take the example of Fig. 1c: specification class `ListSpec` only provides specifications for methods `add` and `of` in `List`. An "attached" specification class may

[4] Intuitively, "aggregable" means that it does not translate to branching instructions in bytecode; see previous work on BYTEBACK for a precise definition [22].

also include additional methods, typically to introduce predicates and other specification elements—such as method `is_mutable()` in Fig. 1c's attached specification class `ListSpec`.

Ghost specifications. *Ghost* code denotes code that is introduced only for the purpose of expressing a specification (or other annotations needed for deductive verification purposes) [9]. The term "ghost" was chosen because it suggests that it could be removed without having any effects on the program's behavior. According to this definition, all `BBlib` annotations are ghost code; however, in this paper we will use the term "ghost specification" in a stricter sense to denote only *attached* specifications. The distinction matters for BYTEBACK because an attached specification, unlike a regular source-level specification, is not processed by the compiler consistently with its intended semantics. In particular, ghost specifications are not propagated by inheritance because the compiler does not know that they refer to methods in a different class from where they are declared. To work around this limitation, we equipped `BBlib` with the `Ghost.of` operator, which provide a means to link elements of ghost specifications introduced in different attached specification classes. Fig. 1d shows examples of using `Ghost.of` to specify the class invariants of `ArrayList` and `LinkedList`: `Ghost.of(ListSpec.class, this).is_mutable()` refers to specification predicate `is_mutable()`, introduced in specification class `ListSpec`, and evaluated on an instance `this` of the specified classes `ArrayList` and `LinkedList`. During analysis, BYTEBACK will first weave `is_mutable()` into `List`, propagate it to its subtypes `ArrayList` and `LinkedList`, and finally rewrite the `Ghost.of` annotation so that it correctly refers to this attached method `is_mutable()`.

```
@Invariant(I_A)
class A {

  @Require(P_A)
  @Ensure(Q_A)
  S m()

}
```

```
@Invariant(j_B)
class B extends A {

  @Override
  @Require(r_B)
  @Ensure(e_B)
  S m()

}
```

```
@InvariantOnly(I_C)
class C extends A {

  @Override
  @RequireOnly(P_C)
  @EnsureOnly(Q_C)
  S m()

}
```

Fig. 3: Three classes `A`, `B`, and `C` related by inheritance, and their specifications.

3.4 Specification Inheritance

Behavioral substitutability constrains how the specification of a method can change between classes that are related by inheritance [20]. Consider two classes `A` and `B` such that `B` is a subclass of `A`; a method `m` is defined in `A` and overridden in `B`; P_A and Q_A denote `A.m`'s pre- and postcondition.

precondition weakening: `B.m`'s precondition P_B must be weaker or as strong as P_A; in formulas: $P_A \implies P_B$.

postcondition strengthening: `B.m`'s postcondition Q_B must be stronger or as strong as Q_A; in formulas: $Q_B \implies Q_A$.

As it is customary in behavioral specification languages [10], BYTEBACK interprets pre- and postcondition annotations in a way that enforces such overriding constraints. Fig. 3 shows classes `A` and `B` as above, whose methods `m` are annotated with `BBlib`'s `@Require` and `@Ensure`, which define the following pre- and postconditions:

- `B.m`'s precondition P_B is the disjunction $P_A \vee r_B$; since $P_A \implies P_A \vee r_B$, this is a weakening of `A.m`'s precondition.
- `B.m`'s postcondition Q_B is the conjunction $Q_A \wedge e_B$; since $Q_A \wedge e_B \implies Q_A$, this is a strengthening of `A.m`'s postcondition.

When more freedom in the definition of the specification of overridden methods is needed, `BBlib` offers annotations `@RequireOnly` and `@EnsureOnly`, also demonstrated in Fig. 3's class `C`: `C.m`'s precondition is just P_C, and `C.m`'s postcondition is just Q_C as declared. When verifying a program annotated with `@RequireOnly` or `@EnsureOnly`, BYTEBACK generates additional explicit verification conditions that check that all the `@RequireOnly`s introduce weaker formulas, and the `@EnsureOnly`s introduce stronger formulas. In Fig. 3's example, BYTEBACK would check that formulas $P_A \implies P_C$ and $Q_C \implies Q_A$ are valid.

Finally, the inheritance of class invariants is consistent with their semantics (discussed in Sec. 3.2): the class invariant I_B of a class `B` inheriting from `A` must be stronger or as strong as `A`'s class invariant I_A. The `@Invariant` annotation enforces this constraint by taking the conjunction of all declared invariants: in Fig. 3, class `B`'s class invariant I_B is $I_A \wedge j_B$, which is a strengthening of I_A. Similarly to pre- and postconditions, annotation `@InvariantOnly` declares a full class invariant explicitly, and checks that it is indeed a strengthening during verification. In Fig. 3, the class invariant of class `C` is literally I_C, but BYTEBACK will check that $I_C \implies I_A$ holds.

3.5 BYTEBACK Verification: Overview

BYTEBACK's verification process inputs a project's bytecode, usually packed in a `.jar` file that includes all compiled source code and `BBlib` annotations in several classes. BYTEBACK queries Soot [13,11]'s *scene* object to extract static information about the bytecode under analysis, and to ultimately translate the input program's semantics into the intermediate verification language Boogie [3]. To this end, BYTEBACK first performs *attaching* (described in Sec. 3.6): it weaves the specification elements supplied in separate classes into the source code they refer to. Then, as outlined in Sec. 3.7, BYTEBACK performs a series of incremental, mostly *local* transformations of each method's bytecode, which explicitly model the program's behavior—such as the modular semantics of method calls, or the exceptional control flow. These transformations work on the Vimp intermediate representation: an extension with specification features of Soot's Jimple readable bytecode representation [22]. Since specifications are provided as `BBlib` annotations (possibly even in separate classes, using the `@Attach` mechanism), they are not available, in general, down the inheritance hierarchy, since the compiler is not aware of their intended semantics. Therefore, BYTEBACK also takes care of explicitly *propagating* specifications according to inheritance from the Vimp methods where they were

explicitly introduced (see Sec. 3.8). After all these transformations, BYTEBACK finally *encodes* the ensemble Vimp components into a *Boogie program* that faithfully encodes the semantics of the input program and its specification. BYTEBACK's output Boogie program can be verified with the Boogie intermediate verifier to determine whether the input JVM program is correct.

```
@Attach(A)
@Invariant(I_ASpec)
abstract class ASpec {

  @Require(P_ASpec)
  @Ensure(Q_ASpec)
  S m() { /* ... */ }

  @Behavior
  abstract boolean f();

}
```

(a) Attached specification class `ASpec`.

```
abstract class A {


  abstract S m();

  @Require(P_A)
  S n() { /* ... */ }


}
```

(b) Partially specified class `A`.

```
@Invariant(I_ASpec)
abstract class A {
  @Require(P_ASpec)
  @Ensure(Q_ASpec)
  S m() { /* ... */ }

  @Require(P_A)
  S n() { /* ... */ }

  @Behavior
  abstract boolean f();
}
```

(c) Class `A` after attaching `ASpec` to it.

Fig. 4: An example of how attaching works in BYTEBACK. Empty padding added for clarity.

3.6 Attaching

By querying Soot's scene for the program under analysis, BYTEBACK retrieves all sorts of information about classes, their inheritance relation, their member declarations and implementations, and so on. BYTEBACK first processes each class S annotated with `@Attach(`C`)`, and copies the specification elements from S to C. If an element specified in S already has a specification in C, the attaching process replaces C's specification with S's. Attaching can also be used to replace *implementations*: if a method is declared as **`abstract`** in C but is given an implementation in S, the attaching process copies this implementation into C. Fig. 4 shows an example of attaching: class `A` in Fig. 4b includes an abstract method `m` and a concrete method `n`; the latter is annotated with a precondition $P_{\texttt{A}}$. The process of attaching Fig. 4a's class `ASpec` to `A` produces what shown in Fig. 4c: now, `A` includes a class invariant and a specification predicate `f`; in contrast, attaching does not modify method `n` or its specification, since `n` is not declared in `ASpec`; finally, method `m` gets the specification declared in `ASpec`, as well as its implementation.

Attaching can also target the parts of a program that are only available as external libraries, without access to their implementation. In such cases, attaching equips a library with a specification, which BYTEBACK then uses to reason about client code that uses that library. For example, Fig. 1d equips library class `ArrayList` with a class invariant; BYTEBACK does not have access to `ArrayList`'s implementation, but can still use this class invariant as a property of every instance of `ArrayList` used in the program under verification.

Fig. 5: BYTEBACK incrementally transform Vimp code to encode the semantics of bytecode instructions and specifications.

3.7 Vimp-level Transformations

After attaching, BYTEBACK performs a series of transformations to all components of the program under verification, incrementally taking care of modeling a different aspect of a program in a form suitable for verification. For example, there are transformations to model the exceptional control flow, to process specification expressions, to translate bytecode instructions, and to handle loop invariants. Fig. 5 outlines the first two local transformations: $\mathcal{T}_{\text{exc}}$ expresses the implicit exceptional control flow as an explicit control flow; $\mathcal{T}_{\text{spec}}$ "aggregates" `BBlib` specification expressions, so that they can be encoded as pure logic expressions suitable for usage as specification constructs.

All these transformations use the Vimp bytecode representation as an intermediate format, which they use to incrementally encode the semantics of the input program and its specification in a form that is amenable to deductive verification. Vimp was introduced in previous work [22]; in the present paper, we further extended it to accommodate a broader range of features that are needed for verification (in particular, a heap model and background axioms). As a result, the final encoding of Vimp into the Boogie intermediate verification language is even simpler than in previous versions of BYTEBACK; relying on Vimp even more extensively also has the advantage that if we need to revise or extend the transformations (for example, to support a new construct with a different semantics), retaining a consistent Boogie encoding would be straightforward in many cases. Thus, we expect that our Vimp revision will also be useful to develop future extensions of BYTEBACK.

Ghost inlining. We introduced two new Vimp transformations to handle some of the features introduced in this paper. One is invariant instantiation, which we describe in Sec. 3.8 because it is applied after specification propagation. The other is ghost inlining, which handles all calls to specification method `Ghost.of` described in Sec. 3.3. For every call `Ghost.of(cls, obj)`—where `cls` is a **`class`** object representing the class where the referenced specification element was declared, and `obj` is a reference to an object to which the specification should be applied—BYTEBACK first checks that class `cls` is attached to a class of type compatible with `obj`'s type. If that's the case, it simply replaces the whole `Ghost.of(cls, obj)` with `obj`. For example, consider again Fig. 1d's class invariant: `Ghost.of(ListSpec.`**`class`**`, `**`this`**`).is_mutable()`; after attaching, this class invariant annotates class `ArrayList`. When it processes this annotation, BYTEBACK checks that `ListSpec` is attached to (i.e., provides a specification for) `List`, and **`this`** is of type `ArrayList`—one of `List`'s subtypes; thus, it simply rewrites the class invariant as **`this`**`.is_mutable()`. Writing directly **`this`**`.is_mutable()` as class invariant of `ArrayList` would have been rejected by the compiler: `is_mutable()` is only available in `List` and its subclasses *after* attaching. Thus, BYTEBACK provides

the dummy method `Ghost.of`, which is accepted by the compiler and then rewritten by ghost inlining.

Algorithm 1 Specification Propagation

```
procedure PROPAGATE(DAG)
    for C ∈ TOPOLOGICALSORT(DAG) do                ▷ For every class C in inheritance order
        for B : C ⟶▷ B do                          ▷ For every direct supertype B of C
            if ¬ONLY(I_C) then                     ▷ If C's invariant was not an @InvariantOnly
                INV(C) ← INV(B) ∧ INV(C)           ▷ Conjoin B's and C's invariants
            for m ∈ OVERRIDES(C) do                ▷ For every method m that C overrides
                if ¬ONLY(P_C.m) then               ▷ If C.m's precondition was not a @RequireOnly
                                                   ▷ Disjoin B.m's and C.m's preconditions
                    PRE(C.m) ← PRE(C.m) ∨ PRE(B.m)
                if ¬ONLY(Q_C.m) then               ▷ If C.m's postcondition was not an @EnsureOnly
                                                   ▷ Conjoin B.m's and C.m's postconditions
                    POST(C.m) ← POST(C.m) ∧ POST(B.m)

procedure TOPOLOGICALSORT(DAG)
    S ← [ ]                                        ▷ S store the list of classes in DAG sorted by subtyping
    for R ∈ ROOTS(DAG) do                          ▷ For every root class R
        P ← [R]                                    ▷ Initialize stack P of classes to process
        while P ≠ ∅ do
            C ← P.POP( )                           ▷ C is the next class to process
            S.APPEND(C)                            ▷ Add C to the end of the sorted list of classes
            for D : D ⟶▷ C do                      ▷ For each direct subtype of C
                                                   ▷ If D is not already sorted, and all direct supertypes of D are already sorted
                if D ∉ S ∧ ∀T : (D ⟶▷ T) ⟹ (T ∈ S) then
                    P.PUSH(D)                      ▷ Process D next
    return S
```

3.8 Specification Propagation

The Vimp transformations outlined in Sec. 3.7 are mostly *local* to each method. Before producing a Boogie program, BYTEBACK needs to *propagate* the specification elements of classes (class invariants) and methods (pre- and postconditions) according to the inheritance hierarchy of the program. In fact, BYTEBACK cannot rely on the compiler to propagate `BBlib` specifications, because specifications are given as `BBlib` annotations, whose semantics the compiler ignores. Besides, even if the compiler could propagate specifications given at the source-code level, it would still not be able to process *attached* specifications, which are declared in a separate class.

Algorithm 1 presents BYTEBACK's specification propagation algorithm. The algorithm's input is a directed acyclic graph (DAG) that represents the overall inheritance hierarchy among classes/types; henceforth, $C \longrightarrow\!\triangleright\ B$ denotes that class C is a child (heir, direct descendant) subclass of B in such graph. Since Java (as well as other JVM languages) supports a form of multiple inheritance via **`interface`**s, inheritance is not simply a tree but a DAG, which may have multiple roots (while all classes in-

herit from `java.lang.Object`, interfaces can only inherit from other interfaces). First, BYTEBACK uses topological sorting to produce a total ordering of all classes that respects the partial inheritance order (procedure TOPOLOGICALSORT in Algorithm 1). Then, the main procedure PROPAGATE goes through the sorted list of classes (from supertypes to subtypes). For every class C, it defines its class invariant as the conjunction of C's explicitly declared class invariant and the invariants of all direct superclasses of C. Similarly, for every method m overridden in C, it defines m's precondition (resp. postcondition) as the disjunction (resp. conjunction) of m's explicitly declared precondition (resp. postcondition) in C and the preconditions (resp. postconditions) of m in all direct superclasses of C. In all these cases, if class C's declared class invariant uses `@InvariantOnly`, or m's declared pre-/postcondition in C uses `@RequireOnly`/`@EnsureOnly`, the specification is not propagated from C's superclasses. Instead, BYTEBACK generates additional side verification conditions (not shown in Algorithm 1), checking that the following implications hold:

$$\textsc{Inv}(C) \implies \bigwedge_{B\colon C \longrightarrow\!\triangleright B} \textsc{Inv}(B) \tag{1}$$

$$\bigvee_{B\colon C \longrightarrow\!\triangleright B} \textsc{Pre}(B.m) \implies \textsc{Pre}(C.m) \tag{2}$$

$$\textsc{Post}(C.m) \implies \bigwedge_{B\colon C \longrightarrow\!\triangleright B} \textsc{Post}(B.m) \tag{3}$$

Invariant instantiation. After specification propagation, BYTEBACK performs an additional Vimp transformation to express the class invariant semantics described in Sec. 3.2; this transformation has to run *after* propagation, because it depends on knowing the full, propagated class invariant of every class. First, every method `m` of class `C` with propagated invariant $I_{\texttt{C}}$ is equipped with a "free" precondition (equivalent to an assumption) $I_{\texttt{C}}$ and (regular) postcondition $I_{\texttt{C}}$; constructors of `C` are also equipped with the latter. Second, every method `c_in(... C a ...)` with an argument `a` of type `C` is equipped with a free precondition $\texttt{a}.I_{\texttt{C}}$ (i.e., `C`'s invariant holds for `a`); and every method `C c_out(...)` that returns an object of type `C` is equipped with a (regular) postcondition $\texttt{result}.I_{\texttt{C}}$ (i.e., `C`'s invariant holds for the returned object). With this encoding, BYTEBACK can reason about every usage of an object of class `C` according to `C`'s invariant.

4 Experiments

This section describes several verification experiments that demonstrate BYTEBACK's capabilities to reason about substitutability properties of Java data structures used by clients in Java and other JVM languages.

4.1 Experiment Description

Each experiment consists of client code that uses data structure libraries, namely implementations of `java.util`'s `List`, `Set`, and `Map` provided by Java 17's JDK or by Google's Guava.[c] In each experiments, BYTEBACK inputs the client code, together with a basic input/output specification of the data structures' APIs—specified using the `@Attach` mechanism described in the paper. The verification goal is checking that the client code uses the data structures' operations consistently with their specified behavior (as well as with the expected substitutability properties).

#	EXPERIMENT	LANG	BYTEBACK	BOOGIE	SOURCE	BOOGIE	MET	CLS	ANNOTATIONS	
			TIME [s]		SIZE [LOC]				B	A
1	Mutable Lists	J 17	0.69	1.20	449	11 374	61	5	33	4
2	Mutable Sets	J 17	0.51	0.89	451	10 193	58	6	30	5
3	Mutable Maps	J 17	0.52	0.88	469	10 430	64	6	31	5
4	Nonnullable Lists	J 17	0.51	0.80	470	9 571	64	6	36	5
5	Nonnullable Sets	J 17	0.50	0.80	405	9 471	54	5	29	4
6	Nonnullable Maps	J 17	0.51	0.89	427	9 551	60	5	30	4
7	Unmodifiable Lists	J 17	0.54	0.77	450	11 399	61	6	35	5
8	Unmodifiable Sets	J 17	0.52	0.76	384	10 554	51	5	28	4
9	Unmodifiable Maps	J 17	0.62	0.87	405	10 632	57	5	29	4
10	Nonresizeable Lists	J 17	0.70	1.11	413	11 826	55	5	32	4
11	Guava Immutable Lists	J 17	0.72	0.96	390	10 004	56	5	31	4
12	Guava Immutable Sets	J 17	0.51	0.72	348	9 471	48	5	25	4
13	Guava Immutable Maps	J 17	0.52	0.73	374	9 375	54	5	26	4
14	Invariant Strenghtening	J 17	0.45	0.63	66	7 592	7	3	4	0
15	Precondition Weakening	J 17	0.53	0.84	87	7 681	15	3	6	0
16	Postcondition Strengthening	J 17	0.46	0.72	91	7 668	15	3	6	0
17	Lists Conversion	S 2.13	0.73	1.37	349	22 564	55	9	23	7
18	Sets Conversion	S 2.13	0.69	1.31	309	20 292	47	9	17	7
19	Maps Conversion	S 2.13	0.70	1.30	331	20 377	53	9	18	7
20	Lists Conversion	K 1.8	0.81	1.15	457	12 187	65	5	37	3
21	Sets Conversion	K 1.8	0.65	1.11	415	12 263	57	5	31	3
22	Maps Conversion	K 1.8	0.53	0.88	413	9 985	59	5	30	3
	total		12.90	20.67	7 953	254 460	120	1 116	567	86
	average		0.59	0.94	362	11 566	51	5	26	4

Table 1: Experiments demonstrating how BYTEBACK can specify and verify substitutability properties of standard software components. Each row corresponds to an EXPERIMENT, consisting of several pieces of client code that use library components that we specified using the features presented in this paper. For each experiment, the table reports the LANGuage (Java, Scala, and Kotlin) of the client code (all non-Java experiments also use JDK 17 data structures, and hence they are multilingual); the wall-clock time (in seconds) taken by BYTEBACK to produce a Boogie program, and by BOOGIE to verify it; 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 (MET) and classes (CLS) that make up the program and its specification; the number P of specification predicates (`@Behavior`), and the number A of attached specification classes (`@Attach`) .

Tab. 1 gives some details about the experiments: out of 22 experiments, 16 are written in Java 17, 3 in Scala 2.13, and 3 in Kotlin 1.8. These are the languages of the verified *client code*, which uses data structure libraries written in Java 17; therefore, the Scala and Kotlin experiments are effectively multi-language, as they involve manipulating Java data structures within a different JVM language. Java experiments 14–16 are different than the others: they are simpler, as they focus on demonstrating BYTEBACK's support for non-inherited class invariants, pre-, and postconditions defined with `@InvariantOnly`, `@RequireOnly`, and `@EnsureOnly` (discussed in Sec. 3.4).

Data structure properties. Here is a summary of the data structure behavioral properties that we specified in our experiments.

Mutability. A collection is *mutable* if it supports adding and removing elements to it, as well as replacing existing elements; `LinkedList` and `ArrayList` are examples of mutable collections. Conversely, a collection is *immutable* if it does not support adding, removing, or replacing its elements; `Collections.unmodifiableList` is

an example of method that returns lists that are immutable. Since adding and removing are *optional* operations of the various collection interfaces, the implementations of methods such as `List.add` in an immutable collection simply throw an `UnsupportedOperation` exception (as demonstrated in Sec. 2's example).

Resizeability. A collection is *nonresizeable* if it does not support adding and removing elements to it; conversely, it is *resizeable* if it does support such operations. A mutable data structure is also resizeable; thus, `LinkedList` and `ArrayList` are also examples of resizeable collections. In contrast, there exist collections that are nonresizeable but are still modifiable: method `Arrays.asList` produces an array-backed list whose elements can be replaced, but not added or removed (because its size is fixed to be that of the backing array).

Nullability. With a little abuse of terminology, we call a collection *nullable* if it can store **`null`** as elements; and *nonnullable* otherwise. Method `List.of` produces lists that are immutable and nonnullable: passing a **`null`** value to this method is accepted by the Java compiler but results in a `NullPointer` exception at runtime.

Java experiments. In the experiments, we equipped the various collection classes with specifications that characterize whether they are mutable, resizeable, and nullable; and their operations with pre- and postconditions that express when they are available (for example, `add` terminates exceptionally in nonresizeable collections) and the properties of the objects they return. These specification rigorously express the information that is available in the natural-language documentation of the corresponding Java libraries, but is not accurately captured by the (public) types in the actual implementations. As we discussed in Sec. 2, equipping these specifications relies on the *attach* mechanism to annotate libraries whose implementation is not directly available to BYTEBACK. With these annotations, BYTEBACK can precisely verify the expected behavior of client code, such as distinguishing between the `List` instances in Fig. 1a—one of them is mutable, the other is not.

Scala and Kotlin experiments. The experiments with Scala and Kotlin client code demonstrate how these fine-grained properties of collections interact with the conversion mechanisms these other JVM languages provide to reuse Java collections. Scala and Kotlin have their own collections libraries, designed based on a detailed type hierarchy that properly distinguishes between mutable, immutable, and other kinds of collections. When converting a Java data structure, there is the problem of choosing the most suitable corresponding Scala or Kotlin type. For performance reasons, the utilities in Scala's `JavaConverters` do not copy the content of a Java data structure into a Scala data structure; instead, they create a *mutable* Scala collection that wraps the Java collection by delegating operations to it. If the underlying Java collection is actually immutable, some operations of the Scala collection will fail with an exception. Short of performing an expensive data-structure copy, the designers of Scala's converters had no way of properly handling such scenarios, given that the information about which Java types correspond to immutable collections is simply not available statically (e.g., `List.of` returns an instance of a private class) and, even if it were, it would be inconsistent with the type hierarchy.

Kotlin's type system also distinguishes between mutable and immutable collections; thus, if we want to explicitly convert a Java collection to a Kotlin collection, we can

choose the most appropriate Kotlin type corresponding to the actual properties of that collection. Even though Kotlin's `List` and `MutableList` types both still represent, at the level of bytecode, objects of type `java.util.List`, the Kotlin compiler can keep track of which operations are allowed on which variant of list. Nevertheless, Kotlin's extensive support for Java interoperability still presents other means of introducing inconsistent behavior when Kotlin code interacts with Java code after compilation. Any Java type can seamlessly be used as a Kotlin type; compatibility rules are designed to be as general as possible—but this generality comes at the expense of soundness in some cases. For example, consider a Java method with an argument `lst` of type `java.util.List` that modifies `lst` (e.g., by removing an element). When the method is called from Kotlin, the compiler accepts any instance of Kotlin's `List` and `MutableList` as actual argument; if we pass an instance of immutable `List`, the method will actually succeed and mutate the list, which is an obvious violation of an immutable list's contract. In all these cases, using BYTEBACK's annotations, we can still reason statically about the precise behavior of different lists or other collections, and hence soundly check that every operation will perform as intended.

4.2 Experiment Results

The experiments ran on a GNU/Linux machine with an Intel Core i9-12950HX CPU (4.9 GHz), running Boogie 2.15.8.0, Z3 4.11.2.0, and Soot 4.3.0. We repeated the execution of the experiments six consecutive times; we report the average wall-clock running time of each experiment over all repetitions except the first one (which we discard to account for possible cold-start delays).

The central columns of Tab. 1 show some statistics about the experiments' results—all of which verified correctly as expected. The encoding time is usually around $2/3$ of a second, and roughly (linearly) proportional to the size of the annotated code; Boogie's verification time is around 1 second per experiment, and also roughly proportional to the number of methods and classes to be verified. The Boogie code produced by BYTEBACK is more than twice larger for the Scala examples (64 lines of Boogie code per line of source code) compared to the Java and Kotlin examples (27 lines of Boogie per line of source); the difference is explained by the numerous class definitions that are used by the Scala runtime: even though BYTEBACK does not process these system classes' implementations, it still needs to reason about their specifications and signatures, and how they are (indirectly) used by the program under verification. In fact, the Scala examples also required a larger number of `@Attach`ed classes to annotate the (implicit) classes used by Scala's converters.

5 Related Work

Empirical evidence of substitutability violations. As mentioned in Sec. 1, substitutability violations in languages like Java are not merely a theoretical possibility. On the one hand, an empirical study of over 200 thousand classes from open-source Java projects found that "almost all" inheritance relations have some subtype use [29]. This evidence confirms that programmers design software that relies on substitutability via subtyping.

On the other hand, there is also abundant, direct or indirect, evidence that substitutability violations occur in practice. An analysis of over 20 million Java classes found

that up to 28% of overridden method may introduce exceptional behavior that is incompatible with substitutability [18]; the same study found similar results when investigating other kinds of effects that break substitutability. Applying random testing techniques to detect substitutability violations [27] found that 30% of the analyzed classes (taken from three popular Java libraries) include crashing substitutes, that is subclasses that may lead to a crash (e.g., with an exception) when they replace an instance of the superclass they inherit from. A recent study of exception preconditions (i.e., preconditions that characterize when a method throws an exception) targeting 46 open-source Java projects and several modules of Java 11's JDK [19] found several instances of methods that unconditionally throw an `UnsupportedOperation` exception, possibly leading to a violation of substitutability.

Static analysis of substitutability. Extended type checkers are tools that enrich a programming language's core type checker with annotations that capture more expressive properties as types. The Checker Framework [25,7][d] augments the Java compiler with several of such extended type annotations, to detect errors such as for null-pointer dereferencing, optional data, out-of-bound index, and so on. While the Checker Framework does not currently include any checker for immutable or nullable collections,[5] it is extensible with new annotations (and some third-party checkers support some form of immutability annotations). The Checker Framework also offers stub files[e]—a mechanism similar to BYTEBACK's `@Attach`—to annotate libraries whose source code is not available. Of course, an extended type checker like the Checker Framework and a deductive verifier like BYTEBACK have distinct applications: a type checker is a lightweight tool, which statically verifies, with limited annotation effort, specific properties at the public interface of clients and suppliers; in contrast, a deductive verifier can, in principle, verify arbitrarily complex behavioral properties of every code feature, but also usually requires a substantial annotation effort and highly trained users. Besides, the Checker Framework is primarily a source-level tool, and works only for Java code, whereas BYTEBACK targets bytecode, and supports multi-language programs.

Other forms of static analysis may also support custom annotations that express immutability or similar properties, providing a similar usability as an extended type checker. For example, the Infer analyzer [4]'s impurity plugin includes an `@Immutable` annotation,[f] which can be used to constrain any kind of object to be immutable.

Deductive verification. A comprehensive behavioral specification language such as JML [14] subsumes the specification features that we introduced in this paper. Therefore, deductive verifiers based on JML, such as OpenJML [6] and KeY [2], can fully reason about behavioral substitutability (and its violations) in a similar fashion as BYTEBACK. As in previous work [23,22], our version of BYTEBACK and verifiers such as KeY and OpenJML offer largely complementary strengths and weaknesses: while the latter support a comprehensive set of specification features, and are much more mature tools, BYTEBACK's strengths follow from its unique angle of targeting bytecode-level verification: it seamlessly supports recent versions of Java (in contrast, OpenJML supports mostly Java features up to version 8, and KeY mostly targets a subset of Java 6), and it can also verify programs written in (subsets of) other JVM languages such as Scala and Kotlin, as well as multilingual programs made of components written in different

[5] Here, "nullable" means that may include **`null`**—see Sec. 4.

languages (such as in Sec. 2's example). The `@Attach` mechanism introduced in the present paper supports adding annotations to an existing component without access to the source code; OpenJML's specification files[g] provide a similar functionality for a source-level verifier.[6] In all, our contributions to BYTEBACK further demonstrate how bytecode-level deductive verification can be useful, but are not meant as a replacement of the full-fledged deductive verifiers for Java out there.

On a more theoretical level, there has been work on modeling and reasoning about notions of substitutability that are more flexible than Liskov and Wing [17]'s. Examples include lazy behavioral subtyping [8] and behavioral interface subtyping [21]. These contributions are usually only demonstrated on a core object-oriented language with minimal features; mainstream programming languages such as Java are designed based on stricter (and simpler) typing rules, which are more accessible to ordinary programmers.

6 Conclusions

In this paper, we described an extension of our BYTEBACK technique [23,22] to reason about behavioral substitutability properties at the level of JVM bytecode. To this end, we equipped BYTEBACK with capabilities to specify and reason about ghost specification predicates, a simple form of class invariants, specification inheritance, and a mechanism to annotate library components indirectly, without access to their source code. Our experiments demonstrate that this extension can precisely verify programs that use "optional" features of widely used Java libraries (such as immutable vs. mutable collections), even in multi-language program where client code and libraries are written in two different JVM languages.

The most natural continuation of this work would be equipping BYTEBACK with a more expressive class invariant specification methodology, which would support reasoning about behavioral substitutability for more complex multi-object structures.

[6] In BYTEBACK, a class specification can be split into multiple `@Attach`ed modules, which in turn can augment (or redefine) an existing source-level specification of the same class; furthermore, BYTEBACK's attach mechanism can also redefine *implementations* of a method in another part of the project under analysis. This makes `@Attach` a somewhat more flexible mechanism than OpenJML's spec files and the Checker Framework's aforementioned stub files, which do not support such a piecemeal specification style and cannot replace implementations.

References

1. Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P.H., Ulbrich, M.: The KeY platform for verification and analysis of Java programs. In: Giannakopoulou, D., Kroening, D. (eds.) Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8471, pp. 55–71. Springer (2014). https://doi.org/10.1007/978-3-319-12154-3_4, https://doi.org/10.1007/978-3-319-12154-3_4
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification—The KeY Book, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6, http://dx.doi.org/10.1007/978-3-319-49812-6
3. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented P rograms. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17, https://doi.org/10.1007/11804192_17
4. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving Fast with Software Verification. In: Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9058, pp. 3–11. Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1, https://doi.org/10.1007/978-3-319-17524-9_1
5. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: Proceedings of CAV. pp. 480–494. Lecture Notes in Computer Science, Springer (2010)
6. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014. EPTCS, vol. 149, pp. 79–92 (2014). https://doi.org/10.4204/EPTCS.149.8, https://doi.org/10.4204/EPTCS.149.8
7. Dietl, W., Dietzel, S., Ernst, M.D., Muslu, K., Schiller, T.W.: Building and using pluggable type-checkers. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. pp. 681–690. ACM (2011). https://doi.org/10.1145/1985793.1985889, https://doi.org/10.1145/1985793.1985889
8. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Lazy behavioral subtyping. J. Log. Algebraic Methods Program. **79**(7), 578–607 (2010). https://doi.org/10.1016/J.JLAP.2010.07.008, https://doi.org/10.1016/j.jlap.2010.07.008
9. Filliâtre, J., Gondelman, L., Paskevich, A.: The spirit of ghost code. Formal Methods Syst. Des. **48**(3), 152–174 (2016). https://doi.org/10.1007/S10703-016-0243-X, https://doi.org/10.1007/s10703-016-0243-x
10. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.J.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3), 16:1–16:58 (2012). https://doi.org/10.1145/2187671.2187678, https://doi.org/10.1145/2187671.2187678
11. Karakaya, K., Schott, S., Klauke, J., Bodden, E., Schmidt, M., Luo, L., He, D.: SootUp: A redesign of the Soot static analysis framework. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference,

TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14570, pp. 229–247. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_13, https://doi.org/10.1007/978-3-031-57246-3_13
12. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Proceedings of FM. pp. 268–283. Lecture Notes in Computer Science, Springer (2006)
13. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop (CETUS 2011) (Oct 2011), https://www.bodden.de/pubs/lblh11soot.pdf
14. Leavens, G.T., Schmitt, P.H., Yi, J.: The Java Modeling Language (JML) (NII shonan meeting 2013-3). NII Shonan Meet. Rep. **2013** (2013), https://shonan.nii.ac.jp/seminars/016/
15. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Proceedings of ECOOP. pp. 491–516. Lecture Notes in Computer Science, Springer (2004)
16. Leino, K.R.M., Schulte, W.: Using history invariants to verify observers. In: Proceedings of ESOP. pp. 80–94. Lecture Notes in Computer Science, Springer (2007)
17. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994). https://doi.org/10.1145/197320.197383, https://doi.org/10.1145/197320.197383
18. Maddox, J., Long, Y., Rajan, H.: Large-scale study of substitutability in the presence of effects. In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (eds.) Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. pp. 528–538. ACM (2018). https://doi.org/10.1145/3236024.3236075, https://doi.org/10.1145/3236024.3236075
19. Marcilio, D., Furia, C.A.: Lightweight precise automatic extraction of exception preconditions in Java methods. Empirical Software Engineering **29**(1), 30 (2024)
20. Meyer, B.: Object-oriented software construction. Prentice Hall, 2nd edn. (1997)
21. Owe, O.: Reasoning about inheritance and unrestricted reuse in object-oriented concurrent systems. In: Ábrahám, E., Huisman, M. (eds.) Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9681, pp. 210–225. Springer (2016). https://doi.org/10.1007/978-3-319-33693-0_14, https://doi.org/10.1007/978-3-319-33693-0_14
22. Paganoni, M., Furia, C.A.: Reasoning about exceptional behavior at the level of Java bytecode. In: Herber, P., Wijs, A., Bonsangue, M.M. (eds.) Proceedings of the 18th International Conference on integrated Formal Methods (iFM). Lecture Notes in Computer Science, vol. 14300, pp. 113–133. Springer (November 2023)
23. Paganoni, M., Furia, C.A.: Verifying functional correctness properties at the level of Java bytecode. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) Proceedings of the 25th International Symposium on Formal Methods (FM). Lecture Notes in Computer Science, vol. 14000, pp. 343–363. Springer (March 2023)
24. Paganoni, M., Furia, C.A.: ByteBack FASE 2025 replication package (Jan 2025). https://doi.org/10.6084/m9.figshare.28166951.v4, https://doi.org/10.6084/m9.figshare.28166951.v4
25. Papi, M.M., Ali, M., Jr., T.L.C., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: Ryder, B.G., Zeller, A. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008. pp. 201–212. ACM (2008). https://doi.org/10.1145/1390630.1390656, https://doi.org/10.1145/1390630.1390656

26. Polikarpova, N., Tschannen, J., Furia, C.A., Meyer, B.: Flexible invariants through semantic collaboration. In: Proceedings of FM. Lecture Notes in Computer Science, vol. 8442, pp. 514–530. Springer (2014)
27. Pradel, M., Gross, T.R.: Automatic testing of sequential and concurrent substitutability. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. pp. 282–291. IEEE Computer Society (2013). https://doi.org/10.1109/ICSE.2013.6606574, https://doi.org/10.1109/ICSE.2013.6606574
28. Summers, A.J., Drossopoulou, S., Müller, P.: The need for flexible object invariants. In: Proceedings of IWACO. pp. 1–9. ACM (2009)
29. Tempero, E.D., Yang, H.Y., Noble, J.: What programmers do with inheritance in Java. In: Castagna, G. (ed.) ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7920, pp. 577–601. Springer (2013). https://doi.org/10.1007/978-3-642-39038-8_24, https://doi.org/10.1007/978-3-642-39038-8_24

URL References

a. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html
b. https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/ImmutableCollections.java
c. https://guava.dev/
d. https://checkerframework.org/
e. https://checkerframework.org/manual/#annotating-libraries
f. https://fbinfer.com/docs/all-issue-types/#modifies_immutable
g. https://www.openjml.org/tutorial/SpecificationFiles