Constrictor: Immutability as a Design Concept

Elad Kinsbruner
Technion, Haifa, Israel

Shachar Itzhaky
Technion, Haifa, Israel

Hila Peleg
Technion, Haifa, Israel

Abstract

Many object-oriented applications in algorithm design rely on objects never changing during their lifetime. This is often tackled by marking object references as read-only, e.g., using the `const` keyword in C++. In other languages like Python or Java where such a concept does not exist, programmers rely on best practices that are entirely unenforced. While reliance on best practices is obviously too permissive, const-checking is too restrictive: it is possible for a method to mutate the internal state while still satisfying the property we expect from an “immutable” object in this setting. We would therefore like to enforce the immutability of an object’s abstract state.

We check an object’s immutability through a view of its abstract state: for instances of an immutable class, the view does not change when running any of the class’s methods, even if some of the internal state does change. If all methods of a class are verified as non-mutating, we can deem the entire class view-immutable. We present an SMT-based algorithm to check view-immutability, and implement it in our linter/verifier, Constrictor.

We evaluate Constrictor on 51 examples of immutability-related design violations. Our evaluation shows that Constrictor is effective at catching a variety of prototypical design violations, and does so in seconds. We also explore Constrictor with two real-world case studies.

2012 ACM Subject Classification Software and its engineering → Software design engineering; Software and its engineering → Software defect analysis

Keywords and phrases Immutability, Design Enforcement, SMT, Liskov Substitution Principle, Object-oriented Programming

Supplementary Material Software (ECOOP 2024 Artifact Evaluation approved artifact): https://doi.org/10.4230/DARTS.10.2.7 [36]

Funding This research was supported by the Israeli Science Foundation (ISF) grants no. 2117/23 and 651/23.

1 Introduction

Object-oriented code routinely manipulates objects and passes around references to them, some of which are stored in other objects. Parts of the code often rely on some object not being changed during its lifetime. This may be in order to uphold some properties as thread safety [30], security [50] and the stability of invariants [31], allow the use of features like interning [11], or improve the readability of the code [24]. Other considerations include information leakage [50] and concurrency [30]. For these reasons, client code may be written under the assumption that objects on which it relies do not change.

1 Corresponding author.
In each of these use cases, the term *immutability* denotes some set of specific assumptions about the object that the use case requires: when used from multiple threads, an object’s fields must be available to read without data races; to be safe to pass into an API, the client programmer wants to know foreign code is not going to break the API’s relied-upon invariants; when used as a key in a hash table, the library assumes that the hash value of the object is going to remain constant. Despite different needs relying on different assumptions, programming practices rely on one of two solutions: (i) documentation-based agreements at the project or language level [44] that delegate all responsibility to human users, or (ii) annotations that can be checked by the compiler or some external tool.

The first option is extremely expressive—as expressive as humans are—but has the obvious downside of the risk of human error. In the scope of checked annotations, some language features provide some steps in this direction: C++’s `const` keyword and Java’s `final` that designate fields and variables as read-only. However, neither of these is a good match for the cases described above: `final` only blocks assignment to a field or variable, but the referenced object can still be mutated via function calls, and `final` does not provide guarantees about object fields unless those happen to be `final` as well.

C++’s `const` is seemingly a better fit, but is still not expressive enough. First, a similar problem to `final` still exists, where a pointer/reference being `const` and its content being immutable are still managed separately (e.g., `const A* const`), and that decision is still left to the programmer. In addition to that, `const` can be used on a specific method, indicating that the method cannot mutate any fields. This does not allow declaring an entire interface as immutable, only certain method; and other methods, particularly ones introduced via inheritance, can mutate any field, including those accessed by `const` methods. The semantics of `const` cannot be used to enforce the property that an interface and all its implementing hierarchy be immutable. In addition to this, for some use-cases it is too restrictive not to be able to assign to *any* field, so C++ also allows marking fields as `mutable` (can be changed even from `const` methods). It is then, again, the user’s responsibility to use this annotation responsibly, and no formal guarantees are provided by the compiler.

**Immutability in class hierarchies.** When provided with an interface or class that is supposed to be immutable, a programmer would like to take advantage of this immutability for purposes of design simplicity or for various optimizations. However, implementing classes and subclasses can introduce unwanted mutations. Languages like Java and C# handle this by marking key library classes (e.g., strings) as `final` or `sealed`. This still does not protect the user from contract mismatches within the library implementation; and, moreover, it precludes legitimate extensions of classes in ways that do not violate the immutability guarantees. This is one of the instances for which the Liskov Substitution Principle (LSP) [39] applies; inheritance as a language mechanism cannot enforce the preservation of properties, and lack of mutations is one such property. The LSP is a *principle*, rather than a *mechanism*, because it is not always possible to distinguish implementations that preserve the properties and ones that do not; and because the properties themselves are often implicit.

Kotlin collections are an interesting example—Figure 1 shows a truncated version of two interfaces, `List` and `MutableList`, from the Kotlin standard library. As summed up by a Google developer [37]:

*MutableList, as the name implies, is a list that has operations to mutate, or change, its contents: add, remove, and replace items. It’s easy to come to the conclusion that the List type must therefore be immutable. That’s not the case. Lists are ‘read-only’, but they may or may not be mutable. [...] The MutableList interface extends the List*
interface List<E> {
    operator fun get(index: Int): E
    fun indexOf(element: E): Int
    operator fun contains(element: E): Boolean
    // truncated
}

interface MutableList<E> : List<E> {
    fun add(element: E): Boolean
    fun remove(element: E): Boolean
    fun clear()  // truncated
}

class Foo(val someList: List<Int>) {
    init {  // called during object construction
        assert(0 in someList)
    }
    fun doStuff() {
        // some stuff
        val idx = someList.indexOf(0)  // implicit assumption:
        // init assert still holds!
        // some more stuff
    }
}

Figure 1 List and MutableList from Kotlin and client code

interface, so it’s very easy to create a list that you can change, but pass it around to
other code so that code can only read it, even as you’re still making changes.

In other words, since MutableList instances are also List instances, the best we can say is that
List does not allow mutation and does not forbid mutation. An understandably-confused
programmer may create an instance of the Foo class (line 17 of Figure 1) using a MutableList,
which will be allowed by the type-checker. The list might at first satisfy the initial assertion,
but the programmer may then clear it before calling doStuff. doStuff relies on the assertion
in the constructor and dereferences a now-empty list, due to the mistaken assumption that
List objects cannot change. Kotlin’s list hierarchy keeps us from taking advantage of the
type checker to enforce our design decisions.

This shows how immutability-related violations of the LSP are particularly insidious.
For this reason, the Scala standard collections library and the Guava libraries for Java fully
separate their mutable collections from the immutable ones [5,7,21].

Object state: concrete vs. abstract. One possible solution is to “freeze” the memory:
create a copy of an object that disallows mutation of all fields. This “freeze” could be shallow
(as C++’s const would create) or deep (essentially an expensive clone). Such a shallow
“freeze” operation exists in languages such as JavaScript [8] and Ruby [4]. Both approaches
have significant disadvantages as mentioned, and are not widespread. Moreover, object fields
are sometimes used for internal bookkeeping in ways that permits—and requires—to update
their values in situations where the object’s content is not conceptually changed. An example
of this can be seen in ImmutableLookupList (Figure 2), where the field lookupCache is used for
memoizing calls to indexOf. While the class indeed mutates this field, it does so in a way
that is non-observable to the user. In such cases, memory freeze is too strong, as it would
disallow these updates. This requires the same kind of escape hatch that mutable provided
for const, which yet again puts the burden on the programmer to decide which fields present
part of the visible state. In some cases, the distinction is not even possible, because a field may produce a visible effect for some, but not all, of the ways in which it can be mutated. For example, in the standard implementation of the union-find data structure [33], some mutations to the pointer structure may cause visible mutation while others are just different ways of expressing the same data.

The problem with ImmutableLookupList is actually a problem with considering lookupCache to be part of the state. It is, of course, part of the concrete state of an ImmutableLookupList object, i.e., it is part of the memory allocated for the object. However, let us consider how ImmutableLookupList looks to an external observer: lookupCache is used in the implementation of the method indexOf, and mutated by it, but this mutation is not observable—through indexOf or any other method of ImmutableLookupList. It is, in other words, an “implementation detail”, never exposed to any client code. It does not impact the abstract state of the object [54]. What we need, therefore, is immutability of the abstract state of the object.

1.1 Our approach: views and view immutability

In order to separate the abstract state from the fields pertaining to internal implementation, we define an object’s view: the set of methods that expose the abstract state to the rest of the system. The guarantee we want, then, is that if the view of an object is immutable, and this property is enforced down the inheritance tree, the immutable hierarchy can safely accommodate mutations of internal state. An enforcement mechanism less rigid than const or frozen objects can allow optimizations like memoization and caching, while disallowing the introduction of visible mutation into the hierarchy.

We define for each object two sets of methods, the set of immutable methods $I$, annotated by the programmer as @immutable, which are methods that do not mutate the abstract state of the object, and the set of view methods $V$, annotated as @viewmethod, whose return values define the object’s abstract state. In the common case, $V \subseteq I$, and so @viewmethod also indicates @immutable (this is not theoretically required, but conserves user effort). Marking the class as @immutable has the same effect as marking each of the class’s methods as @immutable, with one notable distinction: the class annotation is inherited, and applies to all methods of the inherited class, including new methods that were not inherited from its parent class.
We then define the notion of *view-immutability* with regards to the view \( V \) such that when calling any method from \( I \), the object’s internal state may change, but the abstract state exposed by \( V \) does not. While checking this property is not tractable, we show a relaxed property that can be checked, that implies the stronger property under certain conditions.

The notion of view-immutability is meant to be checked in a modular way—there is no need to verify anything regarding the client code, only the data structures themselves. We expect that common data structures in libraries be annotated with `@immutable` as needed, and client code can use these data structures with the desired guarantees.

Our theory is flexible enough to support weaker notions of immutability, e.g., temporary mutability during an init phase [51], or temporary *immutability*, e.g., immutable references in the type system guaranteeing that referenced objects do not change, as in Rust [40].

We implement our approach in a linter/verifier for Python programs named Constrictor.

We translate each class to an SMT encoding using our translating compiler, Py2Smt, then check whether each of the methods in \( I \) are indeed non-mutating.

**Lightweight verification.** Constrictor does not verify the code for correctness; rather, it checks for adherence to design decisions, which is an easier problem. However, it can still fail: Constrictor’s analysis is bounded, and its reliance on SMT inherits the solver’s limitations. Even with these limitations, Constrictor can still act as a contract-checker. This hinges on the fact that immutability violations are usually not bugs but rather unintended violations of conscious design decisions made by different programmers, and as such, they rarely hide from the programmer—or from Constrictor. Empirically, the immutability property depends mostly on the program’s dataflow and not on complex relationships between values. Sometimes there are some correlations that need to be tracked, e.g., in Figure 2 the cache variable’s value is returned to client code, and so needs to be consistent with a real value/index in the list. When the SMT solver returns unknown, there are two options: if Constrictor is run as a verifier, these unknowns will be treated as violations, whereas if it is run as a linter, only violations for which the solver has returned an answer will be displayed to the user.

We evaluate Constrictor on 51 examples of immutability-related design violations. Our evaluation shows that Constrictor is effective at catching a variety of prototypical design violations, and does so in seconds. We also explore Constrictor with two real world case studies, one fixing a design problem in a collections module, and the other introducing memoization into an immutable design pattern. Moreover, we explore human errors that could be made when providing Constrictor with annotations.

**Contributions.** The contributions of this paper are:

▷ A definition of *view immutability*, and a relaxed definition that can be statically checked.
▷ An SMT-based algorithm for checking view immutability.
▷ Py2Smt, a compiler that encodes Python functions for SMT solvers.
▷ Constrictor, a verifier/linter that implements our algorithm for Python programs.
▷ An empirical evaluation of Constrictor and detailed analysis of the results.

---

2 Our replication package is available as a DARTS artifact [36].
Overview

Constrictor is a linter/verifier for Python, so, from now on, the examples will be written in Python. The general concepts are identical and we will be using full type annotations.

We continue our running example that consists of a list library that includes the interface LookupList in Figure 3. This interface only contains methods that allow for the inspection of instances of its implementors. The programmer’s intent was that instances of LookupList should not be mutated (visibly) through their methods. Users of the library rely on this assumption, which until now was only enforced by comments and naming conventions.

The programmer seeks to formalize this assumption: they add the @immutable annotation. Because LookupList functions as an interface, the substitution principle dictates that the @immutable annotation should hold for inheriting classes as well.

Consider two implementors of LookupList (Figure 4). One of them, UpdatingLookupList, violates this assumption by adding methods that mutate the state in a visible way. The other, MemoizingLookupList, also mutates an object field, but does not change the abstract state of the object as observed through the LookupList interface: the field cached is used for memoization: storing index_of’s most recent input/output. Since both classes update data in object fields, the distinction between them is not a simple semantic check.

Our goal is for Constrictor to warn the user about the @immutable annotation’s violation in UpdatingLookupList, and not generate a spurious warning for MemoizingLookupList.

Immutable abstract state. The sense in which we would like LookupList to be immutable is that the return values of “getter” methods, such as __getitem__, do not change after calling any of LookupList’s methods. In this sense, their abstract state is represented by their “observing” methods, whose return values should not change if we wish to consider LookupList an immutable interface.

We call the set of methods representing the abstract state the class’s view: if two objects can be viewed differently through these methods, they definitely do not represent the same conceptual object. Notice that defining the view as just __getitem__ and get_size would be equivalent to defining it to be all three methods of LookupList, because for any implementation upholding the class contract, two instances agreeing on the return values of __getitem__ and get_size for all parameters would also agree on the return values of the other two methods.

The choice of view is akin to defining the abstract object: index_of only exposes the first instance of every value, and different lists that share the locations of duplicate elements—it does not matter which elements as long as they are duplicates—would be equivalent under a view made up of only index_of. Moreover, if LookupList had a contains method returning...
class MemoizingLookupList[E](LookupList):
    cached: Pair[int, E]
data: list[E]
size: int

def index_of(self, element: E) -> int:
    if self.cached.second == element:
        return self.cached.first
    for i in range(self.size):
        if self.data[i] == element:
            self.cached = Pair(i, element)  # mutation!
            return i
    return -1
# truncated
class UpdatingLookupList[E](LookupList):
data: list[E]
size: int

def index_of(self, element: E) -> int:
    for i in range(self.size):
        if self.data[i] == element:
            return i
    return -1
def add(self, element: E):
    self.data.append(element)  # mutation!
    self.size += 1  # mutation!
def remove(self, element: E):
    self.size -= 1  # mutation!
# truncated

Figure 4 Two implementations of the list interface from Figure 3

whether an element is in the list somewhere, then a view comprising only contains would essentially define the abstract object to be equivalent to a set.

It is therefore important to choose a view that represents the intended abstract state for the class. Modeling a list essentially means modeling a partial function mapping indices to elements, which can be achieved with one of the views above. Between equivalent views, choosing the smallest one will reduce the size of formulas generated by Constrictor, which will usually reduce the tool’s run time.

When considering both implementations of LookupList, it appears as though both implementations cause state mutation by changing fields. However, one, MemoizingLookupList, realizes the contract and does not mutate the state visibly, while the other, UpdatingLookupList, mutates the state in a way that can be observed from outside the class.

This motivates us to define view-immutability: a class is view-immutable if calling any of its methods on any instance with any parameters does not affect the return values of any method in the class’s view. This definition allows MemoizingLookupList and rules out UpdatingLookupList.

2.1 Reasoning about view-immutability

In order to verify view-immutability, and know that our assumptions about the abstract state hold, we would need to prove a very strong property: for every state that an object can reach, and for every method \( m \) that we would like to show is immutable, the state of the object before and after calling \( m \) are indistinguishable for any trailing sequence of methods in the
object’s view. In other words, calling \( m \) (or not calling it) does not change the information returned from the object’s view.

In other words, we would be considering two sequences of calls on object \( o \):

\[
\text{init}(\vec{a}); m_1(); \ldots; m_k(); m(m); m_{k+1}(); \ldots; m_{k+n}()
\]

where throughout the sequence, if \( m_i \) is part of the class view, the return value of \( m_i \) is the same. The calls up to \( m_k \) constitute the object’s initialization phase, which defines all the reachable object states. We assume that all methods are deterministic, so the values returned during initialization are trivially equal, and it remains to be checked for \( m_{k+1}, \ldots, m_{k+n} \).

This task is hard to automate because it requires reasoning about unbounded sequences of method calls. At the very least, some user intervention would be needed, in the form of data-structure invariants or other guidance [12,15,28].

**View abstraction.** Our approach is inspired by successful notions from the field of model checking [20]. Instead of tracking sequences of method invocations, we establish an invariant that holds at every step; one “step” being a synchronous method application \( m_i(\vec{a}) \) on two object states \( \sigma_1, \sigma_2 \). The invariant is derived from our notion of view: we assume that the methods in \( V \) represent the abstract state of the object. Therefore we would like to maintain the invariant that the two states are view-equivalent—that is, all the view methods always return equal values when invoked on \( \sigma_1 \) and \( \sigma_2 \). We denote this by \( \sigma_1 \equiv_V \sigma_2 \).

To translate the problem to model checking, object states are modeled as valuations to the object’s fields (with a signature as defined by the respective class declaration). Methods are then represented as transitions between states. We denote the transition from \( \sigma \) to \( \sigma' \) using the method \( m \) as \( \sigma \xrightarrow{m} \sigma' \). The problem is reduced to safety verification with the relational invariant \( \sigma_1 \equiv_V \sigma_2 \).

While this abstraction deliberately omits some internal information about the state, which may introduce spurious warnings, this modeling makes the problem amenable to well-established model-checking techniques based on SMT. We employ a Floyd-style approach: we construct the control-flow graph of each method and then trace all control paths up to some bound. Every program statement is associated with a first-order semantics, which are composed along each path to construct a path transition relation. The transition relation for the method is the disjunction over all of these paths. More details are given in Section 5.

### 2.2 Validation steps

This subsection walks through how **Constrictor** performs the check as explained, using our motivating example *MemoizingLookupList* to illustrate how **Constrictor** is able to show that this class satisfies the \@immutable contract despite benign mutations caused by its methods.

The *LookupList* interface is annotated as \@immutable, indicating all its methods should be non-mutating. The developer of *LookupList* additionally annotates the \_getitem\_ and get_size methods as \@viewmethod, defining the view of the object. The \@viewmethod annotations are inherited by *MemoizingLookupList* along with the \@immutable annotation on the class. Note that the inherited \@immutable annotation on the class requires all of its methods to be non-mutating, including ones that are not inherited from *LookupList*.

This annotated code is the input to **Constrictor**. **Constrictor** first checks that the view of *MemoizingLookupList* is faithful, i.e., can represent the abstract state of the class. It then verifies that all methods marked \@immutable do not affect the values of the view.
Step 1: Encoding to SMT. First, we convert each Python method $m$ to an internal representation describing an approximation of the changes it makes to the object. We denote this the transition relation of the function and label it $\text{TR}_m$.

For example, in the transition relation of `UpdatingLookupList.add`, the assignment of `self.size` on line 28 of Figure 4 is expressed as $\sigma'[\text{size}] = \sigma[\text{size}] + 1$. The method’s transition relation is the composition of the transitions of all statements across all execution paths, in the standard manner.

**Constrictor**’s semantics component is called Py2Smt, and it operates at the method level by enumerating all execution paths up to a bound (this is used, for example, in loops such as the one in Figure 1), collecting path constraints and constructing the composed transition relation $\text{TR}_m$ symbolically for each method $m$. As is usually the case with bounded model checking [16], the computed $\text{TR}_m$ is an approximation.

Step 2: Agreement formula. The transition relations of the view methods are used to compute a set of predicates that check whether two object states are view-equivalent, i.e. agree on the return values of all methods $m \in V$ (with any arguments). These predicates are constructed by considering all possible program states at the end of each method, where the starting states are two given object states $\sigma_1, \sigma_2$, checking whether the return value is equal in both. A program state—unlike an object state—also valuates all the local variables and, in particular, the method’s return value, which we denote $\sigma[\text{returned}]$. We use $\vec{a}$ to denote the method’s call arguments, which occur in $\text{TR}_m$ as free variables.

$$\text{agree}_m(\sigma_1, \sigma_2) \equiv \forall \sigma'_1, \sigma'_2, \vec{a}.
\text{TR}_m[\vec{a}](\sigma_1, \sigma'_1) \land \text{TR}_m[\vec{a}](\sigma_2, \sigma'_2) \rightarrow \sigma'_1[\text{returned}] = \sigma'_2[\text{returned}]$$

View equivalence is expressed symbolically by conjoining over all view methods. In this example, there are two:

$$(\sigma_1 \equiv_V \sigma_2) \equiv \text{agree}._{\text{getitem}}(\sigma_1, \sigma_2) \land \text{agree}._{\text{get_size}}(\sigma_1, \sigma_2)$$

Step 3: View Fidelity. Using the transition relation for all methods and the agreement formula, we compose for each method $m$ the formula for checking the fidelity of the view:

$$\forall \sigma_1, \sigma_2, \sigma'_1, \sigma'_2, \vec{a}.
\sigma_1 \equiv_V \sigma_2 \land \text{TR}_m[\vec{a}](\sigma_1, \sigma'_1) \land \text{TR}_m[\vec{a}](\sigma_2, \sigma'_2) \rightarrow \sigma'_1 \equiv_V \sigma'_2$$

If this formula is found valid for all methods of the class, it means two objects that are visibly indistinguishable remain visibly indistinguishable after any operation. The formula is valid for all four methods of `MemoizedLookupList`, so its view is faithful.

Step 4: View Immutability. Finally, **Constrictor** uses both the transition relation and the view-equivalence relation to construct the immutability check formula for each `@immutable` method: for every object state, executing the checked method on it will not change the view. For `MemoizedLookupList.index_of`, this means:

$$\forall \sigma, \sigma', \text{idx}.
\text{TR}_{\text{index_of}}[\text{idx}](\sigma, \sigma') \rightarrow \sigma \equiv_V \sigma'$$

The formula for `index_of` is valid, and it can be validated by an SMT solver. This verifies that `index_of` is view-immutable over $V$. In contrast, if we try the same with, e.g., `UpdatingLookupList.add`:

$$\forall \sigma, \sigma', \text{el}.
\text{TR}_{\text{add}}[\text{el}](\sigma, \sigma') \rightarrow \sigma \equiv_V \sigma'$$
The formula for \texttt{add} is not valid, and the solver is able to produce a counterexample to this property. For example, if \( \sigma = \{ \text{data} \mapsto [], \text{size} \mapsto 0 \} \), the TR is satisfied by \( \sigma' = \{ \text{data} \mapsto [el], \text{size} \mapsto 1 \} \); but these are not view-equivalent. In particular, \texttt{get\_size()} returns 0 for \( \sigma \), but 1 for \( \sigma' \).

### 3 Definitions

In this section, we define the necessary components for Constrictor’s analysis. Let \( C \) be a class with fields \( F \) and methods \( S \).

- **Definition 1 (Object State).** The object state of an instance of \( C \) is its logical representation: an assignment giving a value for each field in \( F \).

- **Definition 2 (View).** A view of \( C \) is a set of methods \( V \subseteq S \) that describe the abstract state of the class.

  The view will usually contain getters for core fields of the class, while omitting memoization fields, caches and any other data that is not part of the object’s abstract state. While there are usually many options for selecting \( V \), any specific choice is an expression of intent.

- **Definition 3 (Method Term).** A method term \( \tau \) for method \( m \in S \) is an expression \( m(p_1, \ldots, p_k) \) where \( p_1, \ldots, p_k \) are concrete values of the corresponding parameter types. We denote \( T(X) \) for \( X \subseteq S \) to be the set of method terms for all \( m \in X \). We use the shorthand \( T \triangleq T(S) \).

  A method term \( \tau \), when operating on an object state \( \sigma \), has a return value \( (\sigma.\tau) \) and a post-state \( \sigma' \), for which we denote \( \sigma \xrightarrow{\tau} \sigma' \).

  What we actually want is to reason about two objects being indistinguishable in the sense that view methods, which are the representation of the abstract state of the object, cannot tell them apart. If two objects disagree on the values of view method terms, they are clearly not indistinguishable. However, it is possible the objects agree on the values of view method terms, but after applying some method, view methods of the resulting objects will disagree. This can happen for arbitrarily long sequences of methods, motivating the following definition:

- **Definition 4 (Observable Indistinguishability).** Two objects \( \sigma_0, \sigma_2 \) are observably indistinguishable (OI) \( (\sigma_1 \equiv \sigma_2) \) with respect to view \( V \) if for all method terms \( \tau_1, \ldots, \tau_k \), whenever:

  \[
  \sigma_0 \xrightarrow{\tau_1} \sigma_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_k} \sigma_k \\
  \sigma_0' \xrightarrow{\tau_1} \sigma_1' \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_k} \sigma_k'
  \]

  it is the case that \( \sigma_k', \sigma_k \) agree on the values of all view methods from \( V \).

  Now, view immutability just means that method calls leave objects observably indistinguishable from their previous state:

- **Definition 5 (View Immutability).** A method \( m \in S \) is view-immutable with respect to the view \( V \) if:

  \[
  \forall \tau \in T(\{m\}). \ \forall \sigma, \sigma' \in \Sigma. \ \sigma \xrightarrow{\tau} \sigma' \implies \sigma \equiv \sigma'
  \]

  A class \( C \) is view-immutable if all of its methods are view-immutable, including methods in classes that inherit from \( C \).
This definition is hard to check because observable indistinguishability requires checking arbitrarily long sequences of method calls. However, since we expect the values of the view to reflect the full abstract state of the object, we can consider the following, weaker definition:

Definition 6 (View Equivalence). Instances $\sigma_1, \sigma_2$ of class $C$ are view-equivalent ($\sigma_1 \equiv_V \sigma_2$) if they agree on the values of all method terms of view methods:

$$\forall \tau \in T(V), (\sigma_1.\tau) = (\sigma_2.\tau)$$

For this to work, we expect views to be faithful in their representation of the abstract state of the class. A problem arises if there exist two view-equivalent states, and some method term from $T$, such that when applying the term on both states, the resulting states are no longer equivalent. Conceptually, this means that the view must be missing some information, because there exist two objects with the same view, but that are not interchangeable with respect to their subsequent behavior through application of class methods.

This motivates the following definition:

Definition 7 (View Fidelity). The view $V$ is faithful (or exhibits view-fidelity) if for all two objects $\sigma_1, \sigma_2$ and for all method terms $\tau$:

$$\sigma_1 \equiv_V \sigma_2 \land \sigma_1 \xrightarrow{\tau} \sigma'_1 \land \sigma_2 \xrightarrow{\tau} \sigma'_2 \rightarrow (\sigma'_1 \equiv_V \sigma'_2)$$

Actually, if the view is well-behaved (faithful), view equivalence between two objects implies the stronger property of observable indistinguishability.

Theorem 8 (Central Theorem). If $V$ is a faithful view, and $\sigma_1 \equiv_V \sigma_2$, then $\sigma_1 \equiv \sigma_2$.

Proof. By induction on the length of the distinguishing method call sequence, and using view fidelity for the induction step.

Our algorithm will rely on this theorem: we will check view fidelity and the preservation of view equivalence, and this will allow us to deduce observable indistinguishability.

4 Analysis

Our algorithm for checking if class $C$ is view-immutable, shown in Algorithm 1, starts by computing the immutable set and the view set for the class, by using the class annotations:

- @immutable: A method labeled with @immutable must not affect the abstract state of the object; a class labeled as @immutable is a shorthand for labeling all methods as @immutable and all inheriting classes as @immutable.
Algorithm 1: Immutability checking algorithm

procedure CheckClass(C)
Input: A class C
Output: View unfaithful if the class view does not exhibit fidelity, and a mapping of methods to either Violation or No-violation otherwise.

V ← ViewSet(C)
TRs ← \{m ↦ \text{GetTrOfMethod}(C, m) | m ∈ C\}
if not CheckViewFaithful(V, TRs) then
return View unfaithful
Results ← {}
for all m ∈ ImmutableSet(C) do
Σ = MethodStoreSignature(m) \Rightarrow Collect types of fields and local variables
φ ← ∀σ, σ′ : Σ. TRs[m](σ, σ′) \rightarrow \text{Agree}(V, TRs)(σ, σ′)
if CheckSat(¬φ) then
Results[m] = Violation
else
Results[m] = No-violation
return Results

@viewmethod: adds a method to the view set of the object: the set of methods that, if they return the same values on two different objects, we consider them view-equivalent. A @viewmethod annotation also implicitly adds a @immutable annotation to the method. The user should aspire to providing the smallest view set.

These annotations are passed under inheritance.

In Algorithm 1, ImmutableSet(C) returns all methods annotated (directly or via inheritance) as @immutable, and ViewSet(C) returns all methods annotated as @viewmethod.

For each method in the class, the transition relation is computed as a logical predicate between two SMT variable vectors with the appropriate method store signature. The method store signatures are a correspondence between names of memory locations used in methods and their types. In addition, each method store signature contains the special variable returned that represents the return value of the method. We denote this operation GetTrOfMethod, and it is implemented using Py2Smt, as explained in Section 5.

Next, the view fidelity of the full class is checked: the TRs are used to create a formula directly based on the definition of view fidelity, and its validity is checked. We denote this CheckViewFaithful in Algorithm 1. If the view is unfaithful, a meta-warning is issued.

Then, for each method in the immutable set I, the algorithm constructs a formula that searches for a counterexample to the immutability of the method. First, we compute a formula that is satisfied between two states that are view equivalent by using the Agree(V, TRs) function, shown in Algorithm 2, on the set of view methods and their transition relations. Next, we use the result of Agree to construct a formula that is satisfied by states that are not view-equivalent to their sequent states after application of the method. If the formula is satisfiable, then the class is mutable, and this method is a mutator.

This is essentially a reduction of the problem to model checking. Advancements in SMT solver technology can be applied to achieve better performance in our method as well (also see Section 6.6).
Algorithm 2  View equivalence checking algorithm

function Agree(V, TRs)

Input: A set of view methods V and their transition relations

Output: The set’s agree_V predicate

$\Sigma_s \leftarrow \{ f \mapsto \text{MethodStoreSignature}(f) \mid f \in V\}$

return $\lambda \sigma_0, \sigma_1. \wedge_{f \in V} (\forall \sigma'_1, \sigma'_2 : \Sigma_s[f]. (\text{TRs}[f](\sigma_1, \sigma'_1) \wedge \text{TRs}[f](\sigma_2, \sigma'_2)) \rightarrow \sigma'_1[\text{returned}] = \sigma'_2[\text{returned}])$

Strengthening optimization. One optimization that we found useful in our implementation is strengthening the claim and trying to prove TRs[m](σ, σ‘) → (σ = σ‘) instead of TRs[m](σ, σ‘) → (σ ≡ V σ‘) in cases where the SMT solver returned unknown. This is a stronger property that is easier to check and holds in some cases. If that is the case, we can consider the method as a non-violation.

Correctness. The correctness of the algorithm relies on the following claim:

Â Theorem 9 (Algorithm Correctness). If V is a faithful view, and for any method m of the class C:

$\forall \tau \in T\{\{m\}\}. \forall \sigma, \sigma'. \sigma \sim \sigma' \rightarrow \sigma \equiv_V \sigma'$

then the class C is view-immutable w.r.t. the view V.

Proof. Let σ, σ’ be states such that σ \sim m \sigma’. We can deduce that σ \equiv_V σ’. For view immutability, we need to prove that σ \equiv V σ’. We use Theorem 8 and the fidelity of the view V to deduce the desired property.

5 Implementation

In this section we describe implementation details and design choices of Constrictor. Of these, the lion’s share is our compiler, Py2Smt.

Py2Smt. Py2Smt computes the overapproximations of transition relations of functions and the signatures of classes and functions for Constrictor. It is implemented using the Z3 [23] Python API.

Py2Smt creates a CFG for each Python function, and optimizes it in order to reduce graph size and path length. Function calls that have no summary SMT encoding are inlined into the graph, which means recursion is currently not supported. On the resulting graph, each path from the start vertex to the end vertex represents a potential execution path of the function. Py2Smt translates each operation to its SMT encoding, and all paths through the function are joined by a logical OR operation.

This translation depends on finite paths, so loops require special care: when the number of iterations is known at compile time, loops are completely unrolled. Unbounded loops, on the other hand, are unrolled and truncated to a configurable maximum length of program steps, rather than a fixed number of iterations—100 steps in our evaluation—to create finite paths. This creates an underapproximation of the program’s behavior [16].

Moreover, since precise encoding of loops as logical formulas in decidable fragments of first-order logic is fundamentally impossible, Py2Smt currently supports for only in the cases
of \texttt{range} iterations and iterations over lists. These are implemented by (i) utilizing the theory of sequences; and (ii) automatically converting \texttt{for} loops to a \texttt{while}-like form.

Py2Smt supports most built-in types: integers, floats, booleans, strings, lists, and dictionaries. It also supports arbitrary data types represented by classes, as well as generic classes using the bracket syntax from Python 3.12 [1]. Inheritance is also supported. Py2Smt relies on type hints for method signature inference in some cases. These can be provided by the user, or supplied by any type inference tool, such as \texttt{Pytype} [9].

Py2Smt supports reference types and treats class types in the same way as Python—all arguments are passed by reference, except for primitive types.

\textbf{Use of solvers.} Because the formulas created by Constrictor are at times large and complex, and SMT solvers may have different strengths, Constrictor first tries CVC5 [14] and, if it returns \texttt{unknown}, also tries Z3 [23].

Constrictor has two different operation modes, that differ in their behavior in the case that both solvers return \texttt{unknown} both for the original formula and for the heuristically strengthened one. In “linter-mode”, \texttt{unknown} is treated as “no-violation”, while in “verifier-mode”, \texttt{unknown} is treated as “violation”.

\textbf{Unknown view fidelity.} Algorithm 1 starts by attempting to prove view fidelity. Constrictor gives a meta-warning if it detects the view is not faithful, and proceeds if the view is faithful. If it cannot prove either (i.e., the solver returns \texttt{unknown}) it assumes the view is faithful and proceeds. This does not necessarily mean that the algorithm will result in an \texttt{unknown}, since the view fidelity check reasons about methods that the rest of the algorithm disregards.

\section{Evaluation}

Our evaluation is guided by these research questions:

\textbf{RQ1:} Can Constrictor validate a plethora of hierarchy-related design violations, as well as other cases involving immutability violations and non-violations?

\textbf{RQ2:} Can Constrictor validate realistic modules implementing data structures meant to be used in a larger projects?

\textbf{RQ3:} What is the impact of certain types of annotation mistakes on Constrictor?

\subsection{Benchmarks}

We collected 51 benchmarks comprising two sets:

\texttt{Inheritance:} 24 examples of classes in four immutable class hierarchies, including both design violations and non-violations. Violations in this set include adding mutators to an immutable class, overriding immutable methods in a mutating way, defining a view of the object that returns part of the class’s internal state, etc. Some are classic examples of inheritance in object-oriented programming, and others are synthetic, created to measure Constrictor’s performance for different sources of design-related immutability violations.

\texttt{Non-inheritance:} 19 examples of immutability violations and non-violations in cases unrelated to immutable hierarchies. These exercise Constrictor on a wider array of design issues, taken from online tutorials and the official language documentation for C++ [10]. Benchmarks originally in C++ were manually translated to Python and annotated such that every \texttt{const} C++ method is marked as \texttt{@viewmethod}. 
Aspects & Limitations: 8 synthetic benchmarks crafted to demonstrate various aspects and limitations of Constrictor’s technique. The four types of benchmarks in this set explore: 1) loops are unrolled: violations hidden by the unrolling bound; 2) complicated view fidelity checks: views that are not trivially faithful, and the fact that checking view fidelity is separate from immutability violation checks, so the latter can succeed even when the former fails; 3) state space is overapproximated: one benchmark showing how unreachable code can cause a violation due to the overapproximation of object states; and 4) variable types must be explicitly specified: one benchmark showing cases where type inference cannot give an unambiguous answer without user-provided type annotations.

The Inheritance set contains 24 benchmarks, together measuring 778 lines of code (avg 32.4LOC) across 70 methods. The set contains 32 loops. Three methods suffer from intentional annotation mistakes. Six benchmarks contain lists and two contain dictionaries.

The Non-inheritance set contains 19 benchmarks, together measuring 806 lines of code (avg 39LOC) across 66 methods. The set contains 28 loops. One method suffers from intentional annotation mistakes. Eight benchmarks contain lists and three contain dictionaries.

The Aspects & Limitations set contains eight benchmarks, together measuring 248 lines of code and 20 methods.

Each @immutable and @viewmethod method is classified according whether its implementation violates the annotation. Our benchmark suite contains the following composition:

<table>
<thead>
<tr>
<th>Set</th>
<th>non-violations</th>
<th>violations</th>
<th>total classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>12</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>Non-inheritance</td>
<td>10</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>Aspects &amp; Limitations</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

For all experiments, we define precision as the percentage of no violation detections made by the tool that were correct and recall as the percentage of actual non-violations that were correctly flagged as no violation by Constrictor. All detections are at the function level.

All experiments ran on a 2022 MacBook Pro with an M2 processor and 16 GB of RAM.

6.2 RQ1: Design violations

To test RQ1, we ran Constrictor on all three benchmark sets. Constrictor ran on each benchmark separately, without caching the compilation results of Py2Smt. The timeout for Constrictor was set at 10 minutes. We recorded the full runtime of Constrictor for each class, the result of testing view fidelity, and the result of Constrictor for each method in I.

The results for Inheritance are shown in Table 1 and Non-inheritance and Aspects & Limitations in Table 2. The aspect/limitation of each Aspects & Limitations benchmark is denoted by a superscript. Displayed times are an average over 10 runs. The repeated runs did not differ significantly, except for the Graph benchmark (marked with an asterisk in Table 2), which is discussed below. We computed the precision and recall of Constrictor on the Inheritance and Non-inheritance benchmark sets: since many Aspects & Limitations benchmarks are designed to fail, including them does not make sense.

Constrictor checks all benchmarks but one in under 13 seconds, and all but 8 of the benchmarks (84.3%) in under 5 seconds. 46 benchmarks successfully flag all violations and find no spurious violations. One benchmark was marked as unknown by the SMT solver. Constrictor succeeds in checking view fidelity for all benchmarks: all unfaithful views in Table 2 are accurately reported.

Graph is the only benchmark whose runtimes differed significantly across its 10 runs: two runs completed in under 3 seconds, two more runs completed in about 4 seconds, while the
other six completed in about 12 seconds. This discrepancy is due to variations in solver run times; other components of the benchmark’s run time did not change between runs.

Six of 51 benchmarks fail. Of these, 2 are Aspects & Limitations benchmarks designed to fail (of which, one times out), two benchmarks from the Non-inheritance set, and two benchmarks from the Inheritance set. Benchmarks CachedList (Non-inheritance) and MoveToFrontListSet (Inheritance) find a spurious violation by starting at an unreachable state of the object. Benchmark WrongImpMoveFrontListSet (Inheritance) misses a violation because the mutation occurs after the bound for loop unrolling. Benchmark EvilBinarySearchTree (Non-inheritance) was marked as unknown by the SMT solver.

We conclude that Constrictor verifies designs that are view-immutable but do not pass simple C++-style const-checking, and finds design violations where mutation of the abstract state occurs.

6.3 RQ2 - Case Study 1: Kotlin lists

As our first case study, we consider the Kotlin standard library list hierarchy discussed in Section 1, with two implementing classes: the mutation-supporting ArrayList from the standard library, and the fully immutable SmallPersistentVector from the extension library kotlinx.collections.immutable [6]. Figure 6a summarizes the module’s initial hierarchy.

The library’s developer wants to annotate their code for Constrictor. This involves:
### Table 2

<table>
<thead>
<tr>
<th>Class</th>
<th></th>
<th></th>
<th>Fidelity</th>
<th>Violations</th>
<th></th>
<th></th>
<th>exp/act</th>
<th>Found</th>
<th>Success</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BiCounterFirst</td>
<td>2</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>225</td>
</tr>
<tr>
<td>BiCounterSecond</td>
<td>2</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>304</td>
</tr>
<tr>
<td>BinarySearchTree</td>
<td>2</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>9459</td>
</tr>
<tr>
<td>CachedList</td>
<td>1</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>292</td>
</tr>
<tr>
<td>CounterWithAccessCount</td>
<td>2</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>225</td>
</tr>
<tr>
<td>DefaultDict</td>
<td>2</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>261</td>
</tr>
<tr>
<td>EvilBinarySearchTree</td>
<td>2</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>2</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>17346</td>
</tr>
<tr>
<td>EvilUnionFind</td>
<td>1</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>857</td>
</tr>
<tr>
<td>Graph†</td>
<td>2</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>2762</td>
</tr>
<tr>
<td>ImmutablePerson</td>
<td>3</td>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>205</td>
</tr>
<tr>
<td>ImmutableRgb</td>
<td>3</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>9823</td>
</tr>
<tr>
<td>ListWithAccessCount</td>
<td>1</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>12379</td>
</tr>
<tr>
<td>MultiplyingDictionary</td>
<td>1</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6151</td>
</tr>
<tr>
<td>MutablePerson</td>
<td>4</td>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>381</td>
</tr>
<tr>
<td>NumberShuffler</td>
<td>6</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>477</td>
</tr>
<tr>
<td>StringShuffler</td>
<td>2</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>237</td>
</tr>
<tr>
<td>UnionFind</td>
<td>1</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>77</td>
</tr>
<tr>
<td>WrongfullyAnnotatedCachedList</td>
<td>2</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>403</td>
</tr>
<tr>
<td>WrongfullyImplementedCollatz</td>
<td>2</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6823</td>
</tr>
</tbody>
</table>

Precision: 1.00 Recall: 0.90

1. loops are unrolled
2. complicated view fidelity checks
3. state space is overapproximated
4. variable types must be explicitly specified

---

1. Declaring for each class and interface a set of view methods,
2. Annotating some interface with `@immutable`, which will be inherited, and
3. Running `Constrictor` on the classes in the hierarchy.

**Technical setup.** This case study is comprised of three copies of the eight classes in Figure 6a in three copies that are identical except for the location of the `@immutable` annotation, and a fourth version with the nine classes in Figure 6b. The interfaces were taken from the Kotlin standard library and translated verbatim, modeling abstract methods as empty methods (this makes no difference for `Constrictor`).

Kotlin uses Java’s `ArrayList`, which we translated to Python as faithfully as possible: arrays were converted to lists which are used like arrays. Overloaded methods in Java were translated with different method names as Python does not support overloading. We attempted to

---

3. Kotlin’s original hierarchy is taken from [https://kotlinlang.org/docs/collections-overview.html#collection-types](https://kotlinlang.org/docs/collections-overview.html#collection-types)
model as many methods as possible: `trim_to_size`, `ensure_capacity`, `grow`, `get_size`, `is_empty`, `contains`, `index_of`, `last_index_of`, `to_array`, `get_element_data`, `get`, `set`, `add`, `remove`, `remove_at`, `__hash__`, `clear`, `add_all`, `remove_all`, `retain_all`, `iterator`, and `contains_all` are all modeled. Two subsets of its public methods were not modeled: (i) `list_iterator`, `iterator`, `sublist`, `spliterator` because `Py2Smt` does not support internal classes, and (ii) `forEach`, `removeIf`, `sort`, `replaceAll` because `Py2Smt` does not support function objects. `SmallPersistentVector` was similarly translated as faithfully as possible, implementing `_presized_buffer_with`, `get_size`, `add`, `get`, `contains` and `index_of`.

Each of the three copies of the hierarchy in Figure 6a is about 290 lines of code overall. Specifically, our `ArrayList` is 150 lines of code compared to 511 lines of Java, excluding comments and internal classes. The hierarchy in Figure 6b is 296 lines of code and 51 methods overall. Times for all runs of `Constrictor` in this case study are shown in Table 3.

First attempt. The programmer declares `get_size` on `Collection` and `get` on `List` (which inherits the annotation on `get_size`) as view methods. They then try annotating `List` with `@immutable`. After running `Constrictor` on the classes in the hierarchy, `ArrayList` and `SmallPersistentVector`, `Constrictor` will issue a warning on `ArrayList`, which inherits `List`’s `@immutable` annotation but is mutable. `Constrictor` flags `ArrayList`’s `add` method as an immutability violation. Since `SmallPersistentVector` does uphold its inherited `@immutable` annotation, it is not flagged as a violation. The programmer then tries moving the `@immutable` annotation to either the `Iterable` or `Collection` interfaces, getting the same result.

In fact, the only class in Figure 6(a) on which the `@immutable` annotation would not cause a violation flag by `Constrictor` is `SmallPersistentVector`, on which it is useless. Overall, no interface in the hierarchy represents the immutability properties we expect, and `Constrictor` can detect this problem in the hierarchy.
Table 4 Run times for CONSTRICTOR on implementations of a bidirectional tree in Section 6.4.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Run time (ms)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>322</td>
<td>Violation</td>
</tr>
<tr>
<td>Original Red-Green Tree</td>
<td>415</td>
<td>No violation</td>
</tr>
<tr>
<td>Memoized Red-Green Tree</td>
<td>582</td>
<td>No violation</td>
</tr>
</tbody>
</table>

The fix. To fix this issue, the programmer now separates the mutable and immutable hierarchies by creating a new interface: ImmutableList, which extends the List interface (as seen in Figure 6b). Now there is a clear separation between definitely-mutable classes and definitely-immutable classes. The programmer does not need to change the @viewmethod definitions to do so.

The programmer reruns CONSTRICTOR on the class hierarchy as previously described and gets no violation flags. This case study shows how CONSTRICTOR can help developers uphold immutable hierarchy constraints and declare them to their users.

6.4 RQ2 - Case Study 2: Red-Green trees

For our second case study, consider immutable trees with bidirectional references, i.e., both children and parent references. Smith [52] describes the problem: due to the immutability, we need to set the parent and children fields during initialization. However, initializing the tree with a parent field requires building it top-down, and initializing the tree with a children field requires building it bottom-up. These two requirements are contradictory.

Technical setup. We begin with a naïve implementation: a Node class with parent, children, and data fields. The class has get_data, get_parent, get_children and add_child as its methods, and is meant to be constructed top-down, setting the parent field upon construction. After construction, it is now possible to traverse the structure bottom-up and call the add_child method to initialize the children field.

We implemented this class in Python in 14 non-empty lines. We annotated the class as @immutable and annotated the get_data, get_children and get_parent methods as @viewmethod.

As expected, CONSTRICTOR returns a violation on this class, pointing out that add_child visibly mutates the class. The run time of CONSTRICTOR can be found in Table 4.

First attempt. Red-Green trees [38] are a data structure used in the Roslyn compiler for the .NET framework [3]. Red-Green trees solve the problem of bidirectional references by using two separate node objects to represent each tree node: an internal (and possibly mutable) green node and an immutable red node. The red tree serves as an immutable façade; the user never sees the green nodes. A green tree is constructed bottom-up, initializing each green node with its children. The red tree never exists as a tree, but rather red nodes are created on the fly to match each green node whenever the children of a red node are accessed. Since get_children is a computation instead of a getter, a red node can be initialized with just its parent and internal green node and remain entirely immutable.

We translated the version by Smith, converting 38 lines of C# code to 50 lines of Python code. We marked the RedNode class as @immutable, with get_data, get_value, get_children and get_parent as its view. As expected, CONSTRICTOR did not detect a violation in this implementation since it stores nothing and mutates no field, even in a non-observable way.

However, this implementation is very inefficient, as it creates new red nodes representing the children of a given node in every call to get_children. This can cause both direct run
time overhead, and indirect GC overhead caused by the allocation of many small objects, as noted by Lippert [38]. We would like to improve the performance of our implementation.

**The fix.** We now add memoization to our Red-Green tree: the result of the `get_children` method is stored when first called. Since red trees are immutable there is no reason to recompute this field. The new implementation now measures 55 lines. We then ran Constrictor again: Constrictor still did not report a violation, because the mutation of a field within `get_children` is non-visible, preserving view immutability.

This case study shows Constrictor’s utility not in a class hierarchy but rather on validating the implementation of an immutable data structure. Unlike the previous case study, since Red-Green Trees are used as an internal data structure, the @immutable annotation would serve the project developers to ensure no changes made to the red trees break their immutability. The classes from both case studies are part of our artifact [36].

### 6.5 RQ3: Impact of incorrect annotations

In the following small case studies, we set out to explore Constrictor’s behavior in the presence of incorrect annotation by the user. We examined four types of annotation mistakes, relating to the @immutable and @viewmethod annotations: (i) incorrect specification of the class view, (ii) not marking all relevant methods as @immutable, (iii) marking a method as @immutable instead of @viewmethod and vice versa, and (iv) inheritance causing a non-faithful view. Technically, using the correct annotations is the user’s responsibility. We expect Constrictor to behave under incorrect annotation as if the given annotations reflect the user intention. The purpose of this research question is to explore the results in cases that can be a little more error-prone.

#### Incorrect annotation of the view.

Precise view annotations are required to meet the criteria for Theorem 9. Consider the class `ListWithAccessCount` in Figure 7. The view annotations on this class may seem correct to a novice, but the view is too small. The behavior of `get` is undefined for `idx > self.get_size()`, so two `List` objects may agree on the values of `get` for all indices for which it is defined, while still not representing the same list, because they have different sizes. Also, Constrictor issues a fidelity meta-warning on the view in Figure 7.

The user can also incorrectly select a view for `ListWithAccessCount` that is too large: e.g., by adding `get_access_count` to the view. This is wrong for two reasons: (i) marking `get_access_count` as a view method exposes `self.access_count`, which means `get` is now considered to be mutating, and (ii) the @viewmethod annotation also denotes `get_access_count` itself as immutable, but it also mutates `access_count` before returning it. This means it cannot be both immutable and part of the view. Running Constrictor on `ListWithAccessCount` after denoting `get_access_count` as @viewmethod the class is flagged as a violation (time: 404ms). We recall that not all “public” methods are expected to be view methods, only those that define the abstract state of the object—the guarantees mentioned in Section 1 for view-immutability only require that the class is seen as immutable through the view, but return values for other methods may be affected.

#### Not marking a method or class as @immutable.

In this case, Constrictor will simply not check the method or class. Because it only impacts what is checked, not the view that Constrictor uses, this does not affect Constrictor’s performance on other methods/classes.
Marking a method or class as @immutable instead of @viewmethod and vice versa. This is analogous to a view that is too large (using @viewmethod instead of @immutable) or too small (vice versa). For instance, consider the class SettableList in Figure 8, whose view is only the get_size method. The method get is marked as @immutable even though the user probably intended for it to be a part of the view. The result only partially captures the class’s abstract state. In the current state, Constrictor will not flag a violation for set that is marked as @immutable, because the size of the list does not change.

View fidelity under inheritance. The class Collection in Figure 9 represents a collection interface similar to Kotlin’s. By itself, the class and its view, contains, have no issues. However, a programmer extending it may not be aware that adding methods to an inherited class may cause the view inherited from the parent to be unfaithful. When extended, MyCoolCollection’s view is the contains method inherited from Collection.

The programmer adds to MyCoolCollection the method remove_first, which removes one instance of an element given as a parameter to the method. This method causes the view of MyCoolCollection to be unfaithful, despite there being no change in the view set itself: two collections with the same distinct elements would agree on the return value of contains for all arguments, but after running remove_first, a collection with one instance of each element would become empty, while a collection with multiple instances of some elements would remain non-empty.

The solution in this case is to add a get_element_multiplicity method, which would make the view faithful again.
Constrictor: Immutability as a Design Concept

```
class SettableList[E]:
    arr: List[E]

@immutable
def get(self, idx: int):
    return self.arr[idx]

@viewmethod
def get_size(self):
    return len(self.arr)

@immutable
def set(self, idx: int, elem: E):
    self.arr[idx] = elem

@immutable
def add(self, elem: E):
    self.arr.append(elem)

Figure 8 A class with @viewmethod and @immutable swapped
```

```
class Collection[E]:
    @viewmethod
def contains(self, elem: E) -> bool:
        pass

class MyCoolCollection[E](Collection):
    def remove_first(self, elem: E):
        n = self.get_size()
        for i in range(n):
            if self.get(i) == elem:
                self.remove_at(i)

Figure 9 An example where annotation inheritance may cause a view to become unfaithful
```

6.6 Discussion

Our results explore the bounds of Constrictor’s implementation. In this subsection we tie them back to the theoretical aspects of the technique.

Overapproximation and underapproximation. Constrictor can find spurious violations because we are overapproximating the TR in multiple ways, most importantly by considering all possible states of an object, including unreachable states. This means Constrictor can (and does) flag an illegal mutation or leaking of internal state in a benign method when the model found by the solver has an object in such a state.

Constrictor can also miss violations because of its handling of loops via unrolling. Since the loop is unrolled to a fixed, finite depth, it may be truncated too soon, making a real mutation invisible to Constrictor. Additional optimizations to Constrictor, particularly to Py2Smt, or improvements in the SMT solver could allow loops to be unrolled to a greater depth while preserving a reasonable run time. The introduction of loop invariants could help Constrictor find these violations, but annotating loops with invariants would be an unreasonable burden to the user. Integrating loop invariant inference tools [27] may be a reasonable compromise, but is outside the scope of this work.
**View Fidelity.** The correctness of Algorithm 1 fundamentally relies on the class being checked having a faithful view. **Constrictor** can try to return a result even when the view fidelity check fails, but this result is potentially incorrect. Moreover, view fidelity takes into account all class methods, not only those checked by **Constrictor**, causing its check to take a significant portion of **Constrictor**’s runtime. In large projects, it may be useful to manually check view fidelity and configure **Constrictor** to not check fidelity by itself.

**View equivalence is a bisimulation.** View fidelity essentially means that view equivalence forms a bisimulation between two traces representing the sequence of method calls on an object. Checking view immutability then means checking whether the view equivalence bisimulation holds between two traces that are identical except for a single point where they diverge: one trace performs a step and the other performs a no-op. Our algorithm for checking view immutability can then be seen as a special case of the symbolic model checking algorithm for checking bisimulation between the two traces, with view equivalence as the candidate bisimulation relation. Indeed, one modern algorithm for bisimulation checking for infinite state spaces is based on SMT [55]. This increases our confidence in the ability of our method to generalize, and implies that future improvements in bisimulation checking can also be applied to our technique.

**Reliance on type hints.** Some type hints are fundamental to **Constrictor**’s approach, and cannot be fully replaced with type inference. This is because some logical claims are valid in some theories and invalid in others. For example, the benchmark **VariableTypesMatter** from the Aspects & Limitations set contains two methods with the syntactically identical code segment `if self.some == a1 + a2: self.some = a2 + a1`, which is non-mutating if `a1` and `a2` are integers but mutating if they are strings, because integer addition is commutative and string concatenation is not. Type inference is performed in most cases where it is possible. However, as in the above example, the types of parameters cannot be precisely inferred, so type hints are required for function parameters and field types.

**Py2Smt.** **Py2Smt** is expressive, but it has two sets of limitations: (i) unimplemented Python language constructs, e.g., tuples, format strings, and list-, set-, and dictionary-comprehensions, and (ii) language constructs that are not symbolically expressible in SMT, e.g., general `for` loops and full polymorphism. We still support many common special cases, including iteration over lists and `range` objects, which we consider to be the most important cases for `for` loops. Additional work on **Py2Smt** can extend the scope of **Constrictor**.

**Reliance on SMT solvers.** Even when the formula **Py2Smt** encodes is accurate, there is no guarantee an SMT solver will be able to decide it. Some theories, e.g., arrays in cases where the domains and ranges are not disjoint, are simply undecidable. In **Py2Smt**, reference types are represented by using a heap “array”, which is why complex heap-based structures may yield formulas that return *unknown*. Performance on other theories may vary from solver to solver, which is why **Constrictor** tries both CVC5 and Z3. For example, certain formulas in the theory of sequences, which **Py2Smt** uses to encode lists, are not decidable by Z3 but can be decided correctly by CVC5. This affects performance on benchmarks involving lists.

Solvers are not only limited in the types they can represent, but also in the operations on those types. However, in our search for benchmarks we found that most design violations do not involve complex logic as part of the mutation. Therefore, despite the relatively limited expressiveness of SMT solvers, **Constrictor** can be useful in finding design violations.
Solvers are also not a great burden on the performance of Constrictor: across all benchmarks from all three sets, the wait for solver calls is on average 78ms, with the vast majority finishing in under 110ms. This is a small percentage of the runtime of many of the benchmarks, and of it, the majority of the time is spent in proving view fidelity, rather than on the main proof. The rest of Constrictor’s run time is spent on compilation, as well as other tasks (e.g., building the formulas). Only one benchmark (WrongfullyImplementedCollatz from the Non-inheritance set) causes a solver call that takes over 1 second (1.71 seconds). In general, no benchmark reaches the timeout set to the solver (3 seconds). The one timeout in Table 2 times out before the solver is called. Across all benchmarks in all benchmark sets, all solver calls take 13.06 seconds in total.

6.7 Threats to validity

The main threat to validity of this work is that complex, real-world code can be less straightforward to annotate. There may be more than one way to annotate a class, and deciding on its view can itself be a design decision. We attempt to mitigate this threat by introducing RQ3 to demonstrate the effect of using less precise annotations, as an inexperienced programmer might. There are still other ways in which a programmer can incorrectly annotate their code, and they may affect our results.

Moreover, in large, logic-heavy classes, proving view fidelity is more likely to fail because it needs to reason about all methods in the class, not only the @immutable ones. When the solvers return unknown on the fidelity formula, the result of Constrictor may be unsound, requiring user intervention. This may be unsustainable in a large project setting.

7 Related work

Alternate definitions of immutability. The type of immutability most discussed in the literature is reference immutability—non-mutation of an object’s fields through a specific reference [17,29,34,54]. Mutation can also be allowed only in certain contexts [31,45,51]. This contrasts with object immutability [13], objects whose fields cannot be mutated via any reference. Object immutability requires more complex analyses to enforce [42]. Both definitions may or may not be transitive [46,48,49].

Potanin et al. define abstract immutability [19, Section 2.4] that permits “benevolent” side effects, but do not define what these effects can be or how this property is enforced. Eyolfson elaborates on this definition [26], roughly describing a desired solution which does not exist and is similar to view immutability.

Pure functions are functions that do not have any side effects, and only depend on their parameters. This is a very strong form of non-mutability, uncommon in OOP. A less strict form is defined by the JetBrains @Contract(pure) annotation, which indicates that a method does not “affect program state and change the semantics” (but can itself be affected by the state) [2]. Helm et al. [32] and Stewart et al. [53] unify different flavors of side-effect freedom by representing different definitions as a lattice.

Observational purity is a form of purity in which classes can keep and mutate state for their own use, but the mutated state may not leak out of the class. This similar notion to view immutability was introduced by Naumann et al. [43] for the purpose of formal specifications, as (observationally-) pure functions can be used in logical assertions. A method for checking observational purity was introduced in [12], and requires the user to manually supply invariants and specifications for all methods, which is sensible for settings in which writing specifications for all methods is common practice. This is not suitable for
software engineering, because programmers typically do not write logical specifications for their classes. **Constrictor** implicitly defines an invariant by using view methods, which is slightly less expressive but very lightweight in terms of annotation burden.

Coblenz et al. have compiled a comprehensive classification of immutability types [21], which includes most systems mentioned in this section.

**Tools.** A well-known work on enforcing reference immutability is **Javari** [35, 41, 47, 54]. **Javari**’s type system distinguishes *unassignable variables* and *read-only references*. The former is more similar to Java’s `final` keyword, while the latter is introduced as part of the type system similar to C++. Another type system is introduced by Milanova [42] and allows distinguishing “maybe mutable” values from “definitely mutable” values, but makes no distinction between a variable and the value it stores, which may be a reference itself. Zibin et al. introduced a method to enforce object or reference immutability without changing Java’s grammar by using generic type parameters [56]. They allow excluding fields from the abstract state, much like C++’s `mutable` keyword. There is some work on automatic inference of immutability qualifiers. Eyolfson [26, Chapter 4] introduced **Immutability Check**, which automatically infers `const` qualifiers. Eyolfson also introduced a system that automatically checks and sanitizes writes through `const` references [25].

**Applications of immutability.** Immutability can be part of the specification of a method [45]. Even if it is not necessarily part of the required semantics, it can be proven as a lemma in order to support analyses such as alias analysis [22] or flow analysis [50].

In concurrency, immutability is often proved as an auxiliary property to show commutativity of actions [18] (employing a similar SMT-based technique). This is because calling non-mutating operations in any order should result in the same results for each respective called method. Gordon et al. [30] pursue this in the context of reference immutability.

### 8 Conclusion

Objects whose values remain constant are desirable in software design. Current verification solutions are either too restrictive, barring all changes to the object and not just ones reflected in the object’s abstract state, or too permissive, allowing mutations that can be observed. In this work, we presented a new approach centering around the view of an object, which represents its abstract state, and whose values are expected to remain constant.

We introduced the new concept of view-immutability which expresses that the object’s view does not change in an abstract sense. This solution is implemented as a linter verifier, **Constrictor**, using an SMT-based method, which checks that method bodies adhere to denoted immutability constraints.

**Constrictor** successfully detects a variety of design violations, with precision and recall both over 85%. We explored two large realistic case studies of data structures for which we found immutability to be useful, and **Constrictor** is able to validate immutability or report violations. We also explore a set of smaller case studies for **Constrictor**’s behavior with imprecise annotations.

### References

Constrictor: Immutability as a Design Concept


ImmutableCollectionsExplained · google/guava wiki. URL: https://github.com/google/guava/wiki/ImmutableCollectionsExplained.


Mutable and Immutable Collections | Collections | Scala Documentation. URL: https://docs.scala-lang.org/overviews/collections-2.13/overview.html.


