Inference of Resource Management Specifications

NARGES SHADAB, University of California, Riverside, USA
PRITAM GHARAT, Microsoft Research, India
SHREY TIWARI, Microsoft Research, India
MICHAEL D. ERNST, University of Washington, USA
MARTIN KELLOGG, New Jersey Institute of Technology, USA
SHUVENDU K. LAHIRI, Microsoft Research, USA
AKASH LAL, Microsoft Research, India
MANU SRIDHARAN, University of California, Riverside, USA

A resource leak occurs when a program fails to free some finite resource after it is no longer needed. Such leaks are a significant cause of real-world crashes and performance problems. Recent work proposed an approach to prevent resource leaks based on checking *resource management specifications*. A resource management specification expresses how the program allocates resources, passes them around, and releases them; it also tracks the ownership relationship between objects and resources, and aliasing relationships between objects. While this specify-and-verify approach has several advantages compared to prior techniques, the need to manually write annotations presents a significant barrier to its practical adoption.

This paper presents a novel technique to automatically infer a resource management specification for a program, broadening the applicability of specify-and-check verification for resource leaks. Inference in this domain is challenging because resource management specifications differ significantly in nature from the types that most inference techniques target. Further, for practical effectiveness, we desire a technique that can infer the resource management specification intended by the developer, even in cases when the code does not fully adhere to that specification. We address these challenges through a set of inference rules carefully designed to capture real-world coding patterns, yielding an effective fixed-point-based inference algorithm.

We have implemented our inference algorithm in two different systems, targeting programs written in Java and C#. In an experimental evaluation, our technique inferred 85.5% of the annotations that programmers had written manually for the benchmarks. Further, the verifier issued nearly the same rate of false alarms with the manually-written and automatically-inferred annotations.

CCS Concepts: • **Software and its engineering → Software verification**;

Additional Key Words and Phrases: Pluggable type systems, accumulation analysis, static analysis, typestate analysis, resource leaks, specify-and-check, specify-and-verify

ACM Reference Format:
Narges Shadab, Pritam Gharat, Shrey Tiwari, Michael D. Ernst, Martin Kellogg, Shuvendu K. Lahiri, Akash Lal, and Manu Sridharan. 2023. Inference of Resource Management Specifications. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 282 (October 2023), 24 pages. https://doi.org/10.1145/3622858

Authors' addresses: Narges Shadab, nshad001@ucr.edu, University of California, Riverside, Riverside, CA, USA; Pritam Gharat, t-prgharat@microsoft.com, Microsoft Research, Bengaluru, India; Shrey Tiwari, shreymt@gmail.com, Microsoft Research, Bengaluru, India; Michael D. Ernst, mernst@cs.washington.edu, University of Washington, Seattle, WA, USA; Martin Kellogg, martin.kellogg@njit.edu, New Jersey Institute of Technology, Newark, NJ, USA; Shuvendu K. Lahiri, Shuvendu.Lahiri@microsoft.com, Microsoft Research, Redmond, WA, USA; Akash Lal, akashl@microsoft.com, Microsoft Research, Bengaluru, India; Manu Sridharan, manu@cs.ucr.edu, University of California, Riverside, Riverside, CA, USA.

2475-1421/2023/10-ART282
https://doi.org/10.1145/3622858

1 INTRODUCTION

A resource leak occurs when a finite resource managed by the programmer is not released after it is no longer needed, causing the resource to be held indefinitely by the program. For example, resources such as file descriptors, sockets, and database connections must be explicitly released by the programmer. Resource leaks can lead to the depletion of system resources and poor performance, eventually causing the program or the whole system to crash.

Recent work proposed a specify-and-verify approach to find and prevent resource leaks [Kellogg et al. 2021]. This work first requires the programmer to write a *resource management specification* of how the program intends to manage its resources. The specification language includes annotations that indicate which objects control a resource, and lightweight ownership and aliasing properties that track the flow of resources through the program. Given an annotated program, a verification tool then verifies the correctness of the annotations and concludes the absence of leaks given the annotations. The annotations further allow the checker to be fast through modularity and incrementality: it only needs to analyze one method at a time, and after a code change, it only needs to analyze modified methods.

However, there is a substantial barrier to practical adoption of this approach: the programmer must write the resource management specification. Understanding and specifying resource management protocols can be very challenging, especially for large legacy systems.

This paper presents a novel inference technique to automatically discover a resource management specification that can then be used for verification. Automated inference significantly reduces user effort and broadens the applicability of this style of verification.

Inference of resource management specifications poses multiple challenges. These specifications differ from the types or type qualifier properties [Foster et al. 1999] that most existing inference techniques have focused on in the past. Resource management specifications must capture multiple inter-related properties, including resource ownership, which methods release resources, and aliasing relationships. Effective inference of these properties requires a customized algorithm that infers these facts simultaneously. Furthermore, it is desirable to infer the *intended* specification because a program may be buggy, such as releasing a resource along some control-flow paths but not all. The need for *optimistic inference* (inferring specifications that cannot be verified) is atypical for inference techniques.

Our algorithm performs inference based on how the program creates, passes around, and releases resources. The inference is bootstrapped from tool-provided annotations for resource types in the standard libraries (the JDK and the .NET framework) and optional programmer-written annotations. Our algorithm has two phases.

The first phase determines the owned resources for each class by identifying fields whose declared type might need to be released. It also identifies a "disposal method"[1] of a class that releases the owned resources. A method is optimistically inferred to be a disposal method if it releases a resource on some (but not necessarily all) paths, thereby capturing what is typically the intended protocol. With this optimistic technique, the checker will still report an error after inference is completed, but the error is localized to where the likely-intended specification is violated, and hence where the actual code fix is likely to be needed.

The second phase infers all other parts of a resource management specification. This includes optimistic inference of method signatures (ownership of method parameters and returns) and inference of resource alias relationships. When closing any of multiple objects in the program is adequate to release an underlying resource, we refer to those objects as *resource aliases*.

[1]This disposal method is often named `close` (in Java) or `Dispose` (in C#).

This paper formalizes the inference algorithm as a set of inference rules, such that annotations are inferred by applying the rules to a fixed point. To demonstrate the generality and effectiveness of our approach, we implemented two different instantiations of the algorithm. The first implementation is for Java programs, built on the Checker Framework [Papi et al. 2008] that supports type checking, local type inference, and abstract interpretation. The second implementation is for C# programs, built on the CodeQL framework [Microsoft 2023] that provides a Datalog-like declarative code query language for building custom program analyses.

We performed an experimental evaluation on programs that had been manually annotated and verified (in some cases the verification required suppressing false positive warnings). We ran our technique on un-annotated versions of the programs, and it inferred most of the hand-written annotations (85.5%). Further, most remaining verifier warnings are due to true positive issues in the code or verifier imprecision, not missing or incorrectly-inferred annotations.

Contributions. This paper makes the following contributions:

(1) a novel optimistic inference algorithm for resource management specifications, designed to infer specifications likely intended by the developer;
(2) a formalization of the algorithm that is generic across programming languages;
(3) implementations for Java and for C#, demonstrating the generality of our approach;
(4) experiments that show that the approach is effective in practice.

2 BACKGROUND: RESOURCE MANAGEMENT SPECIFICATIONS

Every program must follow the contract that after a resource is acquired, the resource must be released, permitting the runtime or operating system to reuse the resource. We call the method releasing a resource the "disposal method." In Java, the disposal method is often named `close`, while in C# it is often named `Dispose`. It is an obligation on the programmer to call this method on each object managing a resource after the program is done using it.

Section 2.1 reviews the syntax and semantics for the specifications used by the Resource Leak Checker [Kellogg et al. 2021], via which a programmer communicates how the program allocates resources, passes them around, and releases them. Section 2.2 gives an example resource management specification using this specification language, and finally Section 2.3 briefly describes modular verification of these specifications.

2.1 Annotation Syntax and Semantics

A resource management specification is expressed in Java as annotations, which start with an at-sign (`@`). C# uses attributes instead, which are enclosed within square brackets (`[...]`). This section uses the Java syntax [Kellogg et al. 2021]. There are annotations for expressing required and guaranteed calls, lightweight ownership hints, and resource aliasing. For additional details, see the Resource Leak Checker manual [CheckerRLC 2023].

`@MustCall(m)` is a *type qualifier* that modifies a type. The argument `m` is a method that must be called on any value of that type. We call `m` the type's *must-call obligation*. `@MustCall()` (without an annotation argument) means that no method is required to be called. A type qualifier can be written on a type use or a type declaration.

Here is an example of a `@MustCall` annotation on a type *use*:[2]

```
@MustCall("print") Diagnostic explainToUser() { ... }
```

[2]Our code examples use **boldface** for declared identifiers and blue for resource leak specifications.

The return type of `explainToUser` is `@MustCall("print") Diagnostic`, where `@MustCall("print")` is a type qualifier and `Diagnostic` is the Java basetype. The program must call `print` on the returned diagnostic. This annotation ensures that a client does not create a diagnostic but forget to print it.

When written on a type *definition*, a `@MustCall` annotation indicates a method that must be called on every object of that type (including subtypes). For example, the JDK's `java.io.Closeable` interface is specified as:

```
@MustCall("close") // close() must be invoked on every Closeable object
public interface Closeable { ... }
```

This annotation makes writing `Closeable` equivalent to writing `@MustCall("close") Closeable`. That behavior may be overridden: here is an annotation on a type *use* in `java.io.OutputStream`:

```
@MustCall() OutputStream nullOutputStream() { ... }
```

This method returns an `OutputStream` with no must-call obligation, because a null output stream need not be closed.

`@Calls(exprs, methods)` is a method annotation. When written on a method *m*, it means that *m* guarantees that all methods in `methods` are called on all expressions referenced in `exprs`.[3] Here are two examples using `@Calls`:

```
// This method reads the first ("#1") formal parameter (x), then closes it.
@Calls(exprs = "#1", methods = "close")
public int readAndCloseStream(IntStream x) { ... }

// This method guarantees close() is called on two fields
@Calls(exprs={"this.ownedField1", "this.ownedField2"}, methods="close")
public void closeFields() { ... }
```

`@Calls` enables modular verification in cases where a resource is closed in a callee, e.g.:

```
IntStream  y = new IntStream();
// guaranteed to close y, based on the @Calls annotation
readAndCloseStream(y);
```

`@Calls` is also used for verification of "wrapper" types that store resources in fields, to be illustrated in Section 2.2. Prior work referred to `@Calls` as `@EnsuresCalledMethods` [Kellogg et al. 2021].

`@Owning` and `@NotOwning` express a form of lightweight ownership and ownership transfer. When two aliases exist for the same object, these annotations indicate which alias is responsible for fulfilling must-call obligations. Also, an `@Owning` annotation on a field declaration indicates that the enclosing class is responsible for satisfying the field's must-call obligation at the end of its lifecycle (the "resource acquisition is initialization" pattern [Stroustrup 1994, §16.5]). See Section 2.2 for example uses of these annotations.

Unlike full-fledged ownership type systems, as in Clarke et al. [1998] or the Rust programming language [Klabnik and Nichols 2018], lightweight ownership places *no* restrictions on pointer aliasing in the program. `@Owning` and `@NotOwning` annotations serve only as "hints" to the verifier regarding which reference is responsible for closing a resource, and do not impact soundness of verification. Lightweight ownership suffices since we seek only to verify that resources are freed, not to verify the absence of use-after-free or double-free bugs [Kellogg et al. 2021]. By default, formal parameter types are `@NotOwning` and return types are `@Owning`.

[3]The guarantee only holds when *m* terminates normally, without throwing an exception.

```
1  @MustCall("dispose")
2  class MySqlCon {
3    private final @Owning Connection con;
4
5    @ResourceAlias MySqlCon(
6       @ResourceAlias Connection con) {
7      this.con = con;
8    }
9
10   void use() { ... }
11
12   @Calls("this.con", "close")
13   void dispose() {
14     closeCon(this.con);
15   }
16
17   static @Owning Connection createCon() {
18     Connection obj = ...;
19     return obj;
20   }
21   static void useCon(@NotOwning Connection obj)
22     { ... }
23
24   static void closeCon(@Owning Connection obj) {
25     obj.close();
26   }
27 } // end of class MySqlCon
28
29
30 static void client() {
31   Connection con1 = MySqlCon.createCon();
32   MySqlCon mySqlCon1 = new MySqlCon(con1);
33   mySqlCon1.use();
34   if (...)
35     con1.close();
36   else
37     mySqlCon1.dispose();
38 }
```

Fig. 1. Example resource management specifications. The Resource Leak Checker can modularly verify the absence of resource leaks in this code. Given this program without annotations, our resource specification inference can infer all the annotations.

`@ResourceAlias`[4] captures a "resource-aliasing" relationship. Resource aliases are either standard must-aliased references *or* references to distinct objects that manage the same underlying resource [Kellogg et al. 2021]. Fulfilling the must-call obligation of an expression also fulfills the obligation of all of its resource aliases. `@ResourceAlias` annotations specify a resource-alias relationship between a method's return value (or, for a constructor, the newly-allocated object) and one of its parameters. As an example, consider this method in `java.net.Socket`:

```
class Socket {
  @ResourceAlias OutputStream getOutputStream(@ResourceAlias Socket this) { ... }
}
```

The `@ResourceAlias` annotations denote that the `OutputStream` returned by `getOutputStream` is a resource alias of its receiver argument. So, calling `close()` on the returned `OutputStream` is equivalent to calling `close()` on the `Socket`: calling either one is sufficient to release the underlying resource.

2.2 Resource Management Specification Example

Figure 1 shows a class `MySqlCon` annotated with a resource management specification and a client usage that can be verified as correct. The techniques of this paper can automatically infer all the annotations written in the example, which prior work [Kellogg et al. 2021] required a human to provide.

In fig. 1, field `con` (line 3) has qualified type `@MustCall("close") Connection`. It implicitly has the qualifier `@MustCall("close")` because `java.sql.Connection` objects manage a resource, so the `Connection` class is annotated as `@MustCall("close")` in the JDK (that is, in the verifier's standard library model). The `con` field is annotated as `@Owning`: this implies that the enclosing `MySqlCon` class must have a must-call method that satisfies `con`'s must-call obligation. Accordingly, the

[4]In the implementation, this annotation is named `@MustCallAlias`, because it is more general than resources.

`MySqlCon` class is annotated `@MustCall("dispose")` (line 1), and its `dispose` method is annotated `@Calls("this.con", "close")` (line 12), indicating it guarantees `close()` is called on `this.con`.

The `MySqlCon` constructor stores its argument in field `con` (line 7). The `@ResourceAlias` annotations on the constructor (lines 5 and 6) indicate that the argument and the new object are "resource aliases": fulfilling the must-call obligations of one object also fulfills the obligations of the other object. The Resource Leak Checker can validate method `client` (line 30) — that is, prove that `client` releases all resources — because the `@ResourceAlias` annotations show that `con1` and `mySqlCon1` (lines 31–32) are resource aliases.

`@Owning` and `@NotOwning` annotations on parameters and return types indicate which object reference is responsible for fulfilling must-call obligations. Consider the static methods in fig. 1 (starting at line 17). The factory method `createCon` returns an `@Owning Connection`, indicating the caller is responsible for closing it. `useCon` has a `@NotOwning` annotation on its parameter (line 21), indicating it will *not* take responsibility for closing its parameter. `@Owning` is the default for returns, and `@NotOwning` is the default for parameters, so the annotations on lines 17 and 21 are unnecessary; fig. 1 shows them for emphasis. `closeCon` does take ownership of its argument (line 24), and the `closeCon` call on line 14 enables the tool to verify the `@Calls` annotation on line 12.

2.3 Modular Verification

A program annotated with a resource management specification can be *modularly* verified [Kellogg et al. 2021]. The key intuition is that resource leak detection (but not other related problems, such as proving the absence of use-after-free bugs) is an instance of an *accumulation problem* [Kellogg et al. 2022], which is a restricted class of typestate analysis [Strom and Yemini 1986] that admits sound verification even in the presence of arbitrary, untracked aliasing. Instead, a verifier can perform an intra-procedural dataflow analysis, computing for each program point a set of pairs $\langle V, e \rangle$, where e is an expression with a non-empty `@MustCall` obligation, and V is a set of resource-aliased variables referencing e. If a statement s ensures e's `@MustCall` obligation is satisfied via an operation on some variable in V, then $\langle V, e \rangle$ is not propagated to successors of s. If some $\langle V, e \rangle$ reaches a method CFG's exit node, a resource leak warning is reported.

The `@Owning` and `@Calls` annotations of section 2.1 enable modular verification by informing the verifier when a `@MustCall` obligation is satisfied via another method or alias. A `@MustCall("m")` obligation for e is considered satisfied when `m` is directly invoked on e, but also in the following cases (which rely on annotations): e is returned and the method's return type is `@Owning`, e is passed to another method's `@Owning` parameter, e is written to an `@Owning` field, or e is passed to a method whose `@Calls` specification ensures `m` will be called on e. The verifier also uses `@ResourceAlias` annotations on invoked methods to update its set of resource aliases for an expression, improving precision (e.g., when verifying the `client` method in fig. 1, discussed in section 2.2).

A modular verifier must also ensure that all annotations respect standard subtyping rules in the presence of method overriding and other features of real object-oriented languages. E.g., if a supertype method has a `@Calls` annotation, the `@Calls` annotation on any overriding method must be at least as strong, i.e., it should guarantee that at least the same methods are called. Similarly, if a supertype method parameter is `@Owning`, the corresponding parameter in an overriding method must also be `@Owning`, and if a supertype method return type is `@NotOwning`, an overriding method's return type should also be `@NotOwning`. By enforcing standard subtyping rules, modular verification can proceed *without* constructing a call graph: a call site can be analyzed using the declared target, since any overriding method must respect its specification.

Finally, for soundness, all `@Calls`, `@Owning`, and `@ResourceAlias` annotations must be verified, which can also be done modularly. For further details, see Kellogg et al. [2021].

3 INFERENCE

This section presents declarative rules for our inference algorithm and discusses key properties of the algorithm. Section 3.1 provides the intuition behind optimistic inference. Section 3.2 introduces our inference rule syntax and the base program facts it relies on. Section 3.3 gives rules for phase 1 of inference, and Section 3.4 gives rules for phase 2. Finally, Section 3.5 discusses key properties of the inference algorithm.

3.1 Optimistic Inference

Given a program P (possibly partially annotated), the goal of our optimistic inference is to discover a resource management specification that most likely matches the intended specification of the developer. In the case where P is free of leaks, ideally the inferred annotations would allow a modular verification tool to verify P. A classical type inference approach is to recast the type-checking rules as constraints, then solve the constraints. This approach does not work in our context because the verification algorithm is not expressible as a type system (though parts of it are): for example, `@Owning` is not a type qualifier but instead a hint to the verifier. Inference in our context therefore requires a novel algorithm.

Ideally, optimistic inference should infer the programmer's intention even for *buggy* programs that leak resources on some paths. Consider a class `SocketWrapper` wrapping two sockets `socket1` and `socket2`, with the following `cleanup()` method:

```
void cleanup() throws IOException {
  socket1.close();
  socket2.close();
}
```

This method does not necessarily close `socket2`, because `socket1.close()` could throw an exception. An inference technique that accurately reflects the code's behavior would infer only `@Calls("socket1", "close")` for this method, rather than `@Calls({"socket1", "socket2"}, "close")`. The former specification hinders inference in other parts of the program and, when used by a verification tool, leads to confusing false positive alarms at call sites of `cleanup`. By contrast, the latter specification reflects programmer intent, and a verification tool issues an error within `cleanup`, exactly where a programmer needs to fix the bug. The rules described in the next sections employ this optimistic approach.

3.2 Inference Rule Syntax and Base Facts

Our inference rule formalism is language-independent, subsuming object-oriented languages such as C# and Java. For simplicity and without loss of generality, our formalism assumes a name exists for any expression with a non-empty `@MustCall` obligation; this can be satisfied via introduction of temporary variables. In our formalism, every constructor takes one parameter, and every other method takes two parameters: the receiver parameter and one additional formal parameter. Formal parameters are final (unassignable). Our formalism does not include static methods and fields. These restrictions are similar to other formalisms [Igarashi et al. 2001]. Our implementations handle the full C# and Java languages.

Table 1 shows the input facts that represent program constructs. Optimistic inference arises from the fact that our inference uses a may-analysis, but verification performs a must-analysis. The optimism originates from the facts in table 1 that contain "*exists*" and propagates through the inference rules shown in figs. 2 and 3. For instance, $\textsc{Invokes}(s, m, n, r, p)$ checks whether there exists a statement s in method m that invokes method n with receiver r and other parameters p. However, verification requires that such a method n is successfully executed on all paths in m. The

Table 1. The input facts that represent program constructs.

Name	Definition
Field(f, C)	f is a field of class C
Method(m, C)	m is a non-constructor method of class C
AbstractMethod(m, C)	m is an abstract method of class C
Constructor(m, C)	m is a constructor of class C
FieldType(f, T)	field f has basetype T (the basetype elides type qualifiers like `@MustCall`)
ReturnType(m, T)	method m has return basetype T
ParamType(m, T)	method m's parameter has basetype T
Invokes(s, m, n, r, p)	there *exists* statement s in method m that invokes method n with receiver r and other argument p (optimistic)
WritesField(s, m, f, v)	there *exists* statement s in method m that writes variable v to field f of `this` (optimistic)
NotWrittenAfter(f, s, m)	field f is not assigned after statement s in method m. It can be computed from m's control-flow graph.
Returns(s, m, v)	there *exists* statement s in method m that returns v (optimistic)
ThisOrSuperCall(s, m, m', v)	there *exists* statement s in constructor m that is a `this()` or `super()` call invoking constructor m', passing v as the parameter (optimistic)

inference optimistically assumes that if a programmer wrote a disposal method call on one path, the programmer intended to write it on all paths.

Our inference rules, presented in figs. 2 and 3, are written in the style of Datalog rules, though in places they use logical conditions beyond what Datalog can express (our implementations are not based on pure Datalog solvers). The rules are of the form *fact* ← *condition*, . . ., indicating that the fact is inferred when all conditions after the arrow hold. Facts and conditions have parameters that must be matched; the _ parameter always matches.

In figs. 2 and 3, *Annot facts represent annotations inferred by the algorithm. In ParamAnnot(_, _, p) (a formal parameter annotation), p is the name of the method's non-receiver formal parameter. If the input program already contains a partial resource management specification, its annotations can be represented as input facts.

Our inference algorithm proceeds in two phases. The first phase (section 3.3) infers ownership and disposal methods: `@MustCall` annotations for classes, `@Owning` for fields and some method parameters, and `@Calls` for disposal methods. The second phase (section 3.4) infers all remaining annotation types. Two phases are required since the second phase rules rely on the ownership and disposal method annotations inferred by the first phase.

3.3 Phase 1: Ownership and Destructors

Figure 2 gives rules for the first phase of inference. FieldDisposal(f, m_{fd}) (rule ③ in fig. 2) means that m_{fd} is the direct disposal method for the type T of field f; that is, the declaration of T is annotated with `@MustCall(`m_{fd}`)`. Other methods might also be guaranteed to dispose of f, if they are annotated `@Calls(`f, m_{fd}`)`.

Figure 2 gives three rules for inferring a `@Calls(`f, m_{fd}`)` method annotation: when m_{fd} is invoked directly (rule ④), when another method m' with an appropriate `@Calls` annotation is invoked (rule

$\textsc{ClassAnnot}(\texttt{@MustCall}(m_{cd}), C) \leftarrow$ ①
  $\textsc{Method}(m_{cd}, C)$,
  $\neg\textsc{ClassAnnot}(\texttt{@MustCall}(m'_{cd}), C)$,
  $\forall f \in \mathit{OwningFields}(C)$:
    $\textsc{FieldDisposal}(f, m_{fd})$,
    $\textsc{MethodAnnot}(\texttt{@Calls}(f, m_{fd}), m_{cd})$

$\textsc{FieldAnnot}(\texttt{@Owning}, f) \leftarrow$ ②
  $\textsc{Field}(f, C), \textsc{Method}(m, C)$,
  $\textsc{FieldDisposal}(f, m_{fd})$,
  $\textsc{MethodAnnot}(\texttt{@Calls}(f, m_{fd}), m)$

$\textsc{FieldDisposal}(f, m_{fd}) \leftarrow$ ③
  $\textsc{FieldType}(f, T)$,
  $\textsc{ClassAnnot}(\texttt{@MustCall}(m_{fd}), T)$

$\textsc{MethodAnnot}(\texttt{@Calls}(f, m_{fd}), m) \leftarrow$ ④
  $\textsc{FieldDisposal}(f, m_{fd}), \textsc{Invokes}(s, m, m_{fd}, f, \_)$,
  $\textsc{NotWrittenAfter}(f, s, m)$

$\textsc{MethodAnnot}(\texttt{@Calls}(f, m_{fd}), m) \leftarrow$ ⑤
  $\textsc{FieldDisposal}(f, m_{fd})$,
  $\textsc{Invokes}(s, m, m', \texttt{this}, \_)$,
  $\textsc{MethodAnnot}(\texttt{@Calls}(f, m_{fd}), m')$,
  $\textsc{NotWrittenAfter}(f, s, m)$

$\textsc{MethodAnnot}(\texttt{@Calls}(f, m'), m) \leftarrow$ ⑥
  $\textsc{Invokes}(s, m, m', \_, f)$,
  $\textsc{ParamAnnot}(\texttt{@Owning}, m', \_)$,
  $\textsc{NotWrittenAfter}(f, s, m)$

$\textsc{ParamAnnot}(\texttt{@Owning}, m, p) \leftarrow$ ⑦
  $\textsc{ParamType}(m, T)$,
  $\textsc{ClassAnnot}(\texttt{@MustCall}(m_{pd}), T)$,
  $\textsc{Invokes}(s, m, m_{pd}, p, \_)$

$\textsc{ParamAnnot}(\texttt{@Owning}, m, p) \leftarrow$ ⑧
  $\textsc{Invokes}(s, m, m', \_, p)$,
  $\textsc{ParamAnnot}(\texttt{@Owning}, m', \_)$

Fig. 2. Phase 1 of inference: rules for inferring @Calls, @Owning fields and parameters, and @MustCall on classes.

$\textsc{AlwaysWrittenToOwningField}(p, m) \leftarrow$ ⑨
  $\forall \mathit{path} \in \mathit{NormalPaths}(m).\ \exists\, s \in \mathit{path}.$
    $\textsc{ResourceAlias}(s, p, r), \textsc{WritesField}(s, m, f, r)$,
    $\textsc{FieldAnnot}(\texttt{@Owning}, f)$,
    $\textsc{NotWrittenAfter}(f, s, m)$

$\textsc{ParamAnnot}(\texttt{@Owning}, m, p) \leftarrow$ ⑩
  $\textsc{ResourceAlias}(s, p, r), \textsc{Invokes}(s, m, m', \_, r)$,
  $\textsc{ParamAnnot}(\texttt{@Owning}, m', \_)$

$\textsc{ParamAnnot}(\texttt{@Owning}, m, p) \leftarrow$ ⑪
  $\textsc{Constructor}(m, C), |\mathit{OwningFields}(C)| > 1$,
  $\textsc{AlwaysWrittenToOwningField}(p, m)$

$\textsc{ParamAnnot}(\texttt{@Owning}, m, p) \leftarrow$ ⑫
  $\textsc{ParamType}(m, T)$,
  $\textsc{ClassAnnot}(\texttt{@MustCall}(m_{pd}), T)$,
  $\textsc{ResourceAlias}(s, p, r), \textsc{Invokes}(s, m, m_{pd}, r, \_)$

$\textsc{ParamAnnot}(\texttt{@ResourceAlias}, m, p)$, ⑬
$\textsc{ReturnAnnot}(\texttt{@ResourceAlias}, m) \leftarrow$
  $\textsc{Constructor}(m, C), |\mathit{OwningFields}(C)| = 1$,
  $\textsc{AlwaysWrittenToOwningField}(p, m)$

$\textsc{ParamAnnot}(\texttt{@ResourceAlias}, m, p)$, ⑭
$\textsc{ReturnAnnot}(\texttt{@ResourceAlias}, m) \leftarrow$
  $\textsc{Constructor}(m, C), \textsc{ResourceAlias}(s, p, r)$,
  $\textsc{ThisOrSuperCall}(s, m, m', r)$,
  $\textsc{ParamAnnot}(\texttt{@ResourceAlias}, m', \_)$,
  $\textsc{NotWrittenAfter}(s, f, m)$

$\textsc{ParamAnnot}(\texttt{@ResourceAlias}, m, p)$, ⑮
$\textsc{ReturnAnnot}(\texttt{@ResourceAlias}, m) \leftarrow$
  $\textsc{Method}(m, \_)$,
  $\forall \mathit{path} \in \mathit{NormalPaths}(m).\ \exists\, s \in \mathit{path}.$
    $\textsc{ResourceAlias}(s, p, r), \textsc{Returns}(s, m, r)$

$\textsc{ReturnAnnot}(\texttt{@NotOwning}, m) \leftarrow$ ⑯
  $\textsc{Method}(m, C), \textsc{ReturnType}(m, T)$,
  $\textsc{ClassAnnot}(\texttt{@MustCall(\_)}, T)$,
  $\textsc{Field}(f, C), \textsc{Returns}(s, m, f)$

Fig. 3. Phase 2 of inference: rules for inferring `@ResourceAlias`, `@Owning`, and `@NotOwning` parameters.

⑤), and when f is passed to an `@Owning` parameter (rule ⑥). For all three cases, NotWrittenAfter ensures the field is not overwritten after its `@MustCall` method is invoked.

Two rules (⑦ and ⑧) infer a limited set of `@Owning` parameters, one for when the disposal method is directly invoked on a parameter, and one for when a parameter is passed to another method in an `@Owning` position. Rules for the second phase (section 3.4) infer a larger set of `@Owning` parameters, but we found that limited inference of `@Owning` parameters in phase 1 was important for discovering certain class disposal methods.

An `@Owning` annotation is inferred for a field when some method in the enclosing class invokes its `@MustCall` method, as captured by an inferred `@Calls` annotation (rule ②).

The rule for inferring `@MustCall` annotations for a class C (rule ①) is a bit more complex. The rule aims to identify a single instance method m in C that, when called, is guaranteed to satisfy the `@MustCall` obligations of *all* `@Owning` fields in `C`. The rule allows for inferring at most one `@MustCall` annotation per class. Multiple suitable methods could be handled by a `@MustCall` qualifier supporting disjunction, but this is not supported by the current verifiers.

Note that inferring `@MustCall` on classes relies on inference of `@Owning` fields, and inference of `@Owning` fields may rely on inferred `@MustCall` class annotations. This cyclic dependence could lead to problems in cases where a class has multiple `@Owning` fields of a user-defined type, e.g.:

```
class Wrapper {
  @Owning MyResource1 f1;
  @Owning MyResource2 f2;
  ...
}
```

Suppose that `MyResource1` and `MyResource2` each have a class disposal method (non-empty `@MustCall` annotations on the definitions of `MyResource1` and `MyResource2`) that must be discovered by inference. An issue arises if these `@MustCall` annotations are discovered at different times during inference. In such a case, inference may first annotate just one of the fields as `@Owning` and infer a `@MustCall` annotation for the class based just on this field. Then, the discovery of the second `@Owning` field may invalidate the previously-inferred `@MustCall` annotation. An inference engine that allowed for retracting inferred facts could handle this scenario.

This bad case is rare in practice. A class with `@Owning` fields usually has a *single* public method that closes all the fields, thereby excluding the possibility of initially inferring an incorrect `@MustCall` class annotation. In general, a dependence graph between classes could be used to analyze classes in an order that avoids these ordering issues; we do not formalize this extension. Our implementations ensure that, while analyzing class C, all `@Owning` fields within C are discovered before inferring the `@MustCall` annotation for C (using the current `@MustCall` types for other classes).

3.4 Phase 2: Remaining Annotations

Figure 3 gives rules for the second phase of inference, which handles all remaining annotations. The fig. 3 rules for inferring `@ResourceAlias` and `@Owning` annotations rely on $\textit{ResourceAlias}(s, p, r)$ facts. The fact $\textit{ResourceAlias}(s, p, r)$ means that variables p and r are resource aliases at the program point immediately before statement s. Resource aliases can be computed via a straightforward extension to any algorithm for computing must-aliased variables. The three key properties of resource aliases are:

(1) Every variable is always a resource alias of itself.
(2) All must-aliased pointers at a program point are resource aliases.
(3) Given a call $p = m(q)$, if method m is annotated with `@ResourceAlias` on its parameter and return type, p and q are resource aliases at the program point immediately after the call.

Due to property 3, resource aliases must be re-computed as new `@ResourceAlias` annotations are inferred. Many rules in fig. 3 allow for operations to be performed through a resource alias of the parameter; below we simply say "the parameter" to mean the parameter or its resource aliases.

`@ResourceAlias` annotations are only valid in pairs, one on the return of a method and the other on its parameter. Figure 3 has three rules for inferring these `@ResourceAlias` pairs, capturing the different conditions that will allow the annotations to be verified. The first rule (⑬ in fig. 3) infers `@ResourceAlias` on a constructor if its parameter is always written into an `@Owning` field of the class, and if the class has exactly one `@Owning` field. This rule leverages a helper rule AlwaysWrittenToOwningField (rule ⑨), which checks for appropriate field writes on all normal CFG paths through the method, i.e., all paths corresponding to the method exiting without throwing an exception. (Our implementations use standard dataflow analysis techniques rather than enumerating paths.) The second rule (⑭) captures a constructor passing its parameter to another constructor that already has a `@ResourceAlias` parameter. The third rule (⑮) handles a method that always returns (a resource alias of) its parameter.

Figure 3 also gives rules for inferring `@Owning` annotations on parameters. Phase one of inference inferred a limited number of parameter `@Owning` annotations (see rules ⑦ and ⑧ in fig. 2); in phase two, more can be inferred due to use of resource aliases. Two rules (⑩ and ⑫) match the ParamAnnot(`@Owning`) rules of fig. 2, but also allow operations to occur through resource aliases (rule ⑩ matches rule ⑧, and ⑫ matches ⑦). Note that these two rules do *not* require that the operation occur on all paths (that is, they are optimistic); in our experience, an invocation of a `@MustCall` method strongly implies an intent to take ownership, even if the call does not occur on all paths. The final rule (⑪) is similar to the first rule for inferring `@ResourceAlias` (rule ⑬) but handles the case where a class has multiple `@Owning` fields. In the case of a single `@Owning` field, we prefer to infer `@ResourceAlias` on a constructor, since it gives client code the flexibility to finalize either the passed-in resource or the newly-allocated object.

Finally, fig. 3 gives a rule (⑯) for inferring `@NotOwning` annotations. `@NotOwning` is inferred when a method's return type has a non-empty `@MustCall` type and the method acts as a "getter," returning an instance field of the class. In such cases, verifiers cannot reason about callers satisfying the `@MustCall` obligation of the field, so there is no purpose in making the return type `@Owning`.

3.4.1 Example. We illustrate our inference rules and their interactions using the example of fig. 1. Assume the program initially has no annotations. In phase 1, the first rule from fig. 2 for `@Owning` parameters (rule ⑦) infers an `@Owning` annotation for the `closeCon` parameter (line 24), since `closeCon` invokes `close` on its parameter. Given this `@Owning` parameter, the final rule for inferring `@Calls` (rule ⑥) then applies to `dispose`, yielding the `@Calls("this.con", "close")` annotation on line 12. In turn, this `@Calls` annotation enables the rule for inferring `@Owning` on fields (②), yielding the `@Owning` annotation on line 3. Finally, all annotations are in place to infer `@MustCall("dispose")` on the `MySqlCon` class, via rule ① of fig. 2, concluding phase 1.

In phase 2, `@ResourceAlias` annotations are inferred for the constructor of `MySqlCon` (lines 5 and 6), via rule ⑬ of fig. 3, concluding inference for this example. (Recall that the `@Owning` annotation on line 17 and the `@NotOwning` annotation on line 21 are the defaults.) These inferred annotations enable the `client` method in fig. 1 to pass the verifier.

3.4.2 Non-final Owning Fields. The Resource Leak Checker supports another specification annotation: `@CreatesMustCallFor(value)`, which indicates that a method resets the `value` expression's must-call obligations. This annotation can be useful for specifying certain limited usages of non-final `@Owning` fields. As an example, consider a variant of the `MySqlCon` class from fig. 1. In the variant the `con` field is not final:

```
class MySqlCon {
  private @Owning Connection con;
  ... // previous code
  @CreatesMustCallFor("this")
  void reset() {
    if (this.con != null)
      this.con.dispose();
    this.con = createCon();
  }
}
static void client2() {
  Connection con2 = MySqlCon.createCon();
  MySqlCon mySqlCon2 = new MySqlCon(con2);
  mySqlCon2.use();
  mySqlCon2.reset();
  mySqlCon2.use();
  mySqlCon2.dispose();
}
```

Here, the `@CreatesMustCallFor("this")` annotation on `reset` allows the Resource Leak Checker to verify that this code is free of leaks. The checker ensures that any resource stored in `con` is disposed before `con` is overwritten, and that clients are again obligated to dispose of `MySqlCon` objects after `reset` is called.

We implemented inference of `@CreatesMustCallFor`, but then disabled it, due to the fact that only restricted usage patterns can currently be verified. In our experience, verifiable code using `@CreatesMustCallFor` like that shown above is rare; most real-world code with non-final fields either uses more complex protocols or is just buggy. The data in Kellogg et al. [2021] itself showed that the overall impact of `@CreatesMustCallFor` was questionable at best (see Table 3 of Kellogg et al. [2021]).

In our experience, inference of `@CreatesMustCallFor` across large code bases leads to further problems not described by Kellogg et al. [2021]. For soundness, when a method `m` is annotated with `@CreatesMustCallFor`, the annotation must then also appear on all instance methods of the class that transitively invoke `m`, all methods that `m` overrides, and all methods that override `m`. Further, most clients of these methods with inferred `@CreatesMustCallFor` annotations could not be verified, as they did not follow the restricted usage pattern supported by the verifier.

The stringent rules for verifying `@CreatesMustCallFor` annotations should not be surprising, because storing a resource in a non-final `@Owning` field is risky: the field could be overwritten, and the only reference to the resource lost, yielding a leak. This riskiness has made us believe that re-assigning non-final `@Owning` fields is in fact an anti-pattern that should be avoided whenever possible. So, our inference is configured to infer `@Owning` on non-final fields as appropriate, but *not* to infer `@CreatesMustCallFor`. With this configuration, any overwrite of such a field will yield a warning from the checker, encouraging the developer to shift to a different resource management protocol. Section 5 shows that this decision does not significantly hinder the effectiveness of our inference in practice.

3.5 Key Algorithm Properties

Regarding termination, our inference algorithm applies the rules of sections 3.3 and 3.4 until a fixed point is reached. This aspect of the algorithm terminates: the rules only add annotations (never removing them), and there are a finite number of possible annotations that can be inferred. (Annotations like `@Calls` and `@MustCall` are parameterized, but there are a finite number of possible

parameter values.) As noted in section 3.4, *ResourceAlias* facts may need to be re-computed after new `@ResourceAlias` annotations are inferred. Assuming resource aliases are computed using a standard dataflow analysis expressible in the monotone framework [Kam and Ullman 1977] (e.g., the algorithm given in Kellogg et al. [2021]), these re-computations will also terminate, as new `@ResourceAlias` annotations monotonically increase the set of resource aliases after calls.

The rules in figs. 2 and 3 alone do *not* guarantee that inference is deterministic and will produce a unique solution; guaranteeing determinism for our algorithm requires additional side constraints. The only possible non-determinism arises from the rule for inferring `@MustCall` on classes in fig. 2 (rule ①), particularly from cases where there are multiple candidate `@MustCall` methods for a class, and for cases with multiple `@Owning` fields of user-defined type (previously discussed at the end of section 3.3). The former issue can addressed by giving a deterministic rule for choosing the `@MustCall` method from possible candidates, and (as noted in section 3.3) the latter issue can addressed by requiring that classes be processed in order according to their dependencies, breaking cycles arbitrarily but deterministically. Our implementations are deterministic.

Type inference algorithms are often evaluated based on their soundness and completeness. A sound type inference algorithm only infers annotations that are verifiable. As noted earlier, our optimistic inference is deliberately unsound, as we found this necessary to best capture the specifications intended by developers. However, the combination of optimistic inference plus a sound verification tool is sound in the following way: if after inference, the verifier reports no warnings, the program is guaranteed to be free of resource leaks. This is the same guarantee that the verification tool offers to a human annotator.

A type inference algorithm is complete if it is guaranteed to discover a set of annotations that would make an input program type check, if such a set exists. Unfortunately, there are certain cases where our algorithm is incomplete. One case involves types with multiple `@Owning` fields, like the following:

```
class Wrapper implements Closeable {
  final Socket s1;
  final Socket s2;
  Wrapper(Socket s1, Socket s2) {
    this.s1 = s1; this.s2 = s2;
  }
  ...
}
```

The problem is a lack of expressivity in the resource management specification language: there is no way to express that either the `Wrapper` or *both* the wrapped `Sockets` must be closed. Assuming `Wrapper` has a `@MustCall` method that closes both of the fields, our inference will make both of the fields and both the constructor parameters `@Owning`. However, consider the following client code (exception handling elided):

```
Socket s1 = ..., s2 = ...;
Wrapper w = new Wrapper(s1,s2);
s1.close(); s2.close();
```

Since `Wrapper`'s constructor parameters were inferred to be `@Owning`, client code *must* release those `Sockets` by invoking `Wrapper.close()` to pass the verifier. The verifier will warn about not closing `w` in the code above, even though there is no leak. We did not encounter code of this type in our experiments. If the `Wrapper` type above had a single `Socket` field, our algorithm would infer `@ResourceAlias` annotations instead, which allow for either the `Wrapper` or the wrapped `Socket` to be closed.

Similarly, our algorithm may be incomplete if there are multiple valid `@MustCall` methods for a class, but the algorithm chooses the wrong one for the class annotation. We did observe this to occur rarely in our benchmarks; see discussion of fig. 4 in section 5. As discussed in section 3.3, a `@MustCall` annotation supporting disjunction could solve this issue.

4 IMPLEMENTATION

To demonstrate the generality of our approach, we developed two separate implementations of our inference algorithm. The first implementation works for Java programs, producing annotations that are compatible with the original Resource Leak Checker [Kellogg et al. 2021]. The second implementation targets C# programs and generates annotations suitable for use with RLC# [Gharat et al. 2023], an independent implementation of resource management verification for C#. We describe these implementations in turn.

4.1 Java Inference

The Resource Leak Checker [Kellogg et al. 2021] is built using the Checker Framework, a framework for building pluggable type systems and abstract interpretations (dataflow analyses) [Papi et al. 2008]. Our Java inference implementation is also built on the Checker Framework, leveraging its whole-program inference (WPI) infrastructure [CheckerWPI 2023; Kellogg et al. 2023]. WPI infers type qualifiers by repeatedly running a checker over the input code, using facts it derives to insert new qualifiers on each run, until a fixed point is reached. This built-in WPI functionality cannot infer resource management specifications, as the annotations are mostly not type qualifiers (see section 1).

Our implementation re-uses the WPI fixed-point infrastructure, running alongside the original Resource Leak Checker. After the Resource Leak Checker runs on each method, inference runs as a post-analysis pass, applying the rules of section 3 to discover new annotations. Inference re-uses intermediate results computed by the checker, in particular its computation of resource aliases (see section 3.4); this re-use saves computation and guarantees results consistent with the checker. After each iteration, any newly-inferred annotations are persisted into specification files, and these persisted annotations are visible to subsequent iterations. In our experiments, the algorithm converges after an average of six iterations.

Our current implementation is inefficient, in that it re-analyzes all the program code in each top-level iteration. Using a worklist to only re-analyze necessary methods and classes would be more efficient, but the Checker Framework does not support it. We plan to optimize our implementation in the future (borrowing techniques from our C# inference implementation; see section 4.2). But, since we expect inference to be run infrequently, the speed of the current implementation is not a critical concern.

Our implementation is currently undergoing code review so that it can be incorporated into the Checker Framework. A future release of the framework will include it.

4.2 C# Inference

RLC# [Gharat et al. 2023] is a resource leak checker for C#, built using CodeQL [Microsoft 2023]. Whereas RLC can be viewed as solving an accumulation-based problem, RLC# can be viewed as solving a reachability-based problem — a significantly different design approach. RLC# uses the local data flow engine of CodeQL [CodeQLCSharpDataflow 2023] for intra-procedural analysis, and it uses the specification language of section 2.1 for inter-procedural reasoning (via C# attributes rather than Java annotations).

There are two major language-dependent differences between RLC and RLC#:

Table 2. The portion of hand-written annotations that our algorithm inferred. The "MWA" column gives the total number of manually-written annotations for each benchmark.

		@Owning			@Must-Call-Alias	@Must-Call on class	@Calls	@Not-Owning	Total
	MWA	final fields	non-final fields	params					
Service 1	21	5/5	2/2	0/0	0/0	8/8	6/6	0/0	100%
Service 2	28	3/3	5/5	1/1	2/2	8/8	8/8	1/1	100%
Service 3	24	7/7	1/1	0/0	0/0	8/8	8/8	0/0	100%
Lucene.Net	63	7/7	7/7	13/13	6/6	13/14	12/13	2/3	95%
EF Core	25	1/2	3/3	0/1	2/6	5/6	5/6	1/1	68%
zookeeper	93	6/6	12/16	3/6	18/20	12/25	7/12	8/8	71%
hadoop-hdfs	91	14/17	3/3	10/12	16/23	16/18	2/7	8/11	76%
Hbase	35	7/7	1/1	3/3	0/2	7/11	4/6	4/5	74%
Total	-	93%	89%	83%	75%	79%	76%	83%	-

- Java supports the concept of *checked* and *unchecked* exceptions, whereas C# only has unchecked exceptions. Both RLC and RLC# handle unchecked exceptions unsoundly. This does not impact Java applications because all the critical exceptions in Java are checked. However, the impact is significant in C# applications.
- For *generic types*, Java supports type erasure which is not supported by C#. As a result, C# inference must explicitly associate annotations with each bound for the type parameters. This makes adding annotations in the source code for generic types difficult. We avoid this issue by adding annotations as logical formulae inside the CodeQL query instead of the source code. The logical formula identifies the location and the program element in the source where we need to add an annotation. Annotations as logical formulae avoids repetitive building of code and creating a CodeQL database with every addition of a new annotation.

We implemented our inference algorithm using the same infrastructure as that of RLC#. The inference rules described in section 3 are expressed as a custom query in the CodeQL query language. Then, CodeQL manages the inference fixed point computation internally. As the CodeQL fixed point engine is highly tuned, this inference implementation is much faster than our current Java implementation. Finally, the CodeQL query generates a CSV file containing all inferred annotations. Note that unlike the Java implementation, our C# inference does not repeatedly run the RLC# verifier; instead, relevant logic is shared at the CodeQL query level as needed. This strategy also provides a performance boost, as certain expensive verification computations need not be run during inference.

5 EVALUATION

This section presents an experimental evaluation of our two inference implementations. Our evaluation aims to answer these research questions:

- **RQ1:** How effective is inference in recovering annotations that were previously added manually?
- **RQ2:** How effective is inference in exposing true positive bugs (resource leaks) related to both library types and user-defined types?
- **RQ3:** After running inference, what percentage of the verifier warnings relate to missing or incorrect annotations?
- **RQ4:** What is the running time of inference?

To answer these questions, we ran inference on a suite of Java and C# benchmarks. The results show that our inference technique is effective and makes the specify-and-check approach for resource management verification more practical.

5.1 Recovering Manual Annotations

5.1.1 Benchmarks. To answer RQ1, we collected a suite of Java and C# benchmarks that had been manually annotated to pass the Resource Leak Checker and RLC#, respectively. For Java, we re-used the three large benchmarks from Kellogg et al. [2021], as their artifact provided annotated versions of these benchmarks. The benchmark sizes are shown in table 4; we ran on the exact same modules used in Kellogg et al. [2021]. We updated the annotations as needed to work with the most recent version of the Resource Leak Checker.

For C#, we selected as benchmarks three proprietary microservices (referred to as Service 1, Service 2, and Service 3), and also two open-source projects, Lucene.NET and EF Core. Lucene.NET [Lucene.NET 2023] is a port of the Lucene search library to C#. EF Core [EF Core 2023] is an object-database mapper that works with a variety of backend databases through a plugin API. Benchmark sizes are given in table 4. We manually added annotations to these benchmarks to provide a baseline for comparison with our inference result.

5.1.2 Methodology. For the Java benchmarks, we utilized the manually annotated version provided by Kellogg et al. [2021]. Our inference process does not rely on the presence of manual annotations. Therefore, we removed annotations that were used by the verifier and conducted inference on the unannotated versions of the benchmarks. Subsequently, we calculated the number of manually-written annotations that were successfully identified through the inference process.

For the C# benchmarks, we performed inference on the unannotated versions of the benchmark. Subsequently, we calculated the number of manually-written annotations that were successfully recovered through the inference process.

5.1.3 Results. Table 2 shows the percentage of manually-written annotations that were discovered by our inference algorithm, broken down by each type of annotation. On average, it recovered 85.5% of manual annotations, with 73.7% recovered on average for open-source Java projects, 92.6% on average for open-source C# projects, and 100% for proprietary C# microservices. We hypothesize that our technique is more effective on the microservices because those programs were written under a stricter coding discipline, with more careful review and standards, to prevent leaks in production services.

Note that there were 54 hand-written `@CreatesMustCallFor` annotations that are excluded from table 2, as we found inference to be more effective when inference of `@CreatesMustCallFor` was disabled, as discussed in section 3.4.2. If included, the average percentage of recovered annotations is reduced to 77% from 85.5%; 100% of annotations are still inferred in the proprietary C# services.

Here we present some examples from the Java benchmarks where our inference failed to infer a handwritten annotation. Figure 4 shows a simplified example from Zookeeper where inference missed `@MustCall("shutdown")` on the `Learner` class. Three methods of the `Learner` class satisfy the constraints defined in section 3.2 for being a disposal method for a class. Without information on how `Learner` instances are used, it is difficult to determine which method is the true disposal method. In this case, inference added `@MustCall("closeSockSync")` to the class instead of the desired `@MustCall("shutdown")`. We believe that a better design for this class would have made the `closeSockSync` and `closeSocket` methods private, in which case inference would have added the correct annotation.

Figure 5 gives another example where our inference fails to infer manually-written annotations. The `BlockChecksumComputer` class from Hadoop contains two private resource fields. However, its

```
@MustCall("shutdown") // hand-written
@MustCall("closeSockSync") // inferred
public class Learner {

  protected @Owning Socket sock;

  ...

  @Calls(value="this.sock", methods="close")
  void closeSockSync() {
    try {
      if (sock != null) {
        sock.close();
        sock = null;
      }
      ...
    } catch (IOException e) {
        ...
    }
  }
```

```
  @Calls(value="this.sock", methods="close")
  void closeSocket() {
    if (sock != null) {
      if (...) {
        if (closeSocketAsync) {
          ...
        } else {
          closeSockSync();
        }
      }
    }
  }

  @Calls(value="this.sock", methods="close")
  public void shutdown() {
    ...
    closeSocket();
    ...
  }
}
```

Fig. 4. Simplified example from Zookeeper that shows (1) the complexity of inferring the correct `@MustCall` annotation, (2) the benefits of our optimistic analysis to infer correct annotations that captures programmers' intention.

```
@MustCall("compute") // not inferred
abstract static class BlockChecksumComputer {

  // @Owning annotations that were
  // not inferred by our analysis
  private final @Owning
     LengthInputStream metadataIn;
  private final @Owning
     DataInputStream checksumIn;
  ...

  @NotOwning
  LengthInputStream getMetadataIn() {
      return metadataIn;
  }

  @NotOwning
  DataInputStream getChecksumIn() {
    return checksumIn;
  }

  @Calls(
    value={"this.checksumIn", "this.metadataIn"},
    methods={"close"}) // not inferred
  abstract void compute() throws IOException;
}
```

```
class ReplicatedBlockChecksumComputer
  extends BlockChecksumComputer {

  ...

  @Calls(
    value={"this.checksumIn", "this.metadataIn"},
    methods={"close"}) // not inferred
  void compute() throws IOException {
    try {
      ...
    } finally {
      IOUtils.closeStream(getChecksumIn());
      IOUtils.closeStream(getMetadataIn());
    }
  }
}
```

Fig. 5. Simplified example from Hadoop to illustrate missed annotations and an anti-pattern.

disposal method is defined as abstract, and delegates the responsibility of closing these resources to

Table 3. Inferred resource management specifications and causes for checker warnings. A false positive is correct code that the verifier cannot prove safe, even after annotations are added; the table categorizes this separately from checker warnings that can be eliminated by adding an annotation. The sum of all percentages in each row adds to 100%. "@CMCF" indicates the percentage of warnings reported due to missing `@CreatesMustCallFor` annotations for non-final/non-readonly fields. "@MCE" (MustCall Empty) represents the percentage of warnings reported due to missing `@MustCall()` annotations on class declarations or type uses that do not retain a resource.

	#warnings (no annos)	#warnings (inferred annos)	true positives	false positives	incorrect annotations	missing annotations		
						@CMCF	@MCE	Other
Service 1	251	240	12%	43%	0%	8%	32%	5%
Service 2	45	34	29%	36%	0%	3%	23%	9%
Service 3	20	12	50%	50%	0%	0%	0%	0%
Lucene.Net	670	592	30%	42%	2%	3%	20%	3%
EF Core	88	154	22%	60%	0%	8%	5%	5%
zookeeper	138	170	19%	49%	5%	13.5%	6%	7.5%
hadoop-hdfs	26	95	18%	56%	7%	9.5%	9.5%	0%
Hbase	828	844	19%	44%	2%	7%	9%	19%

the sub-classes that do not have direct access to the variables. This design choice can be problematic not only for the modular verifiers and inference but also for developers, as it can introduce leaks if programmers do not release the resources properly. Our inference does not detect patterns where the resources for `@Owning` fields are only released in subclasses, so it misses the the `@Calls` annotation on `compute`, the `@Owning` annotations on the fields, and the `@MustCall("compute")` annotation on the class.

5.2 Impact on Verifier Warnings

For RQ2 and RQ3, we ran the verifier on two versions of each benchmark, first with no annotations, and second with the annotations inferred by our algorithm. Then, for the checker warnings reported after inferring annotations, we categorized them by whether they were caused by incorrect inferred annotations, missed annotations, resource leaks (true positive bugs), or false positive warnings from the checker. A true positive is a real resource leak, while a false positive is correct code that the verifier cannot prove safe, even after annotations are added. As there were too many warnings to triage all of them, we randomly chose at least 50 warnings from each benchmark (or all warnings if the total number of warnings is less than 50) to categorize. Table 3 shows the results.

Number of Warnings. Per table 3, inference sometimes increases the number of warnings reported by the verifier, compared to an unannotated program. For resource leaks, this result is expected, since our inference discovers new `@MustCall` obligations that the verifier does not check if the code is unannotated. Consider the simple example in fig. 6, where instance field `con` contains a Java `Connection`. For unannotated code (left), the checker reports a single warning that this object is being written into a non-`@Owning` field. With inferred annotations (right), `con`'s inferred `@Owning` annotation leads to inferring a `@MustCall` obligation for the `ConnectionWrapper` class. If in multiple places, client code uses `ConnectionWrapper` objects without calling its disposal method, multiple warnings are then reported, an increase over the unannotated program. But, the warnings after inference are of a higher quality, since they better reflect the intended resource management

```
public class ConnectionWrapper {
  private final Connection con;
  public ConnectionWrapper() {
    // warning: assign to non-@Owning field
    this.con = new Connection(...);
  }

  public void close() {
    this.con.close();
  }
}
...
// no warnings here
ConnectionWrapper cw = new ConnectionWrapper();
ConnectionWrapper cw2 = new ConnectionWrapper();
...
... no calls to close ...
```

```
@MustCall("close")
public class ConnectionWrapper {
  private final @Owning Connection con;
  public ConnectionWrapper() {
    // no warnings here
    this.con = new Connection(...);
  }
  @Calls(value="this.con", methods="close")
  public void close() {
    this.con.close();
  }
}
...
// two warnings: disposal method not called
ConnectionWrapper cw = new ConnectionWrapper();
ConnectionWrapper cw2 = new ConnectionWrapper();
...
... no calls to close ...
```

Fig. 6. Location of verifier warnings before (left) and after (right) inference.

```
@Calls(value = {"this.storage", "this.committedTxnId", "this.curSegment"}, methods = {"close"})
public void close() throws IOException {
  storage.close();
  // committedTxnId remains open if storage.close() throws an exception
  IOUtils.closeStream(committedTxnId);
  IOUtils.closeStream(curSegment);
}
```

Fig. 7. Simplified example from Hadoop that shows the benefit of optimistic inference.

specification of the program. For this reason, we focus our evaluation on assessing the quality of warnings reported after inference.

Impact of Optimistic Inference. During manual triage, we found multiple cases where optimistic inference provided better results than a technique that only infers verifiable annotations. Consider the `shutdown` method in fig. 4, which serves as the disposal method of the class and calls `closeSocket`. There are some paths in `closeSocket` in which the `sock` object is either null or already closed, and hence `close` is not called. However, our algorithm still optimistically annotates the `closeSocket` method with the `@Calls("this.sock", "close")` annotation, matching the code's intention to guarantee that the `Socket` is closed. This annotation is subsequently used by the checker to verify the `@Calls("this.sock", "close")` annotation on the `shutdown` method.

Another example is shown in fig. 7, where `committedTxnId` and `curSegment` point to resource objects that remain open on the possible exceptional exit caused by the `storage.close()` call. However, our inference algorithm is able to infer the `@Calls({"this.storage", "this.curSegment", "this.committedTxnId"}, {"close"})` annotation. This annotation leads to an error report within the `close()` method, the location that actually requires a fix.

True and False Positives. As shown in table 3, the verifier reports an average of 28% and 19% (out of total warnings) true positives in C# and Java benchmarks respectively after inference runs. This true positive rate is very close to the average 26% precision reported for the Resource Leak Checker [Kellogg et al. 2021], where precision is the ratio of true positive warnings to all tool warnings. However, that work reported precision *after* laborious work to manually annotate the

programs. The fact that we achieve a similar precision rate shows that our inference is highly effective. Also, Kellogg et al. [2021] reports statistics only about library types, due to the voluminous number of errors reported and the difficulties of manual annotation. Our statistics cover (a random sample of) all errors: both library and user-defined types. While investigating the random sample, we discovered 6 true positive leaks of user-defined resources in Java projects, which had been ingored in the previous work. We also discovered 10 more true positives for user-defined types in C# benchmarks that were not reported by RLC# with manual annotations.

The Resource Leak Checker and RLC# report a significant number of false positives on our benchmarks, 46% and 48% of the total warnings for C# and Java benchmarks respectively. These tools have a high false positive rate because they are sound and do not use heuristics to filter warnings. Still, previous work showed that the false-positive rate from the RLC was comparable to that of heuristic (unsound) checkers [Kellogg et al. 2021]. The two major reasons we observed for false positives were (a) conservative handling of collection types such as lists and dictionaries (i.e., poor handling of generics), and (b) a lack of path sensitivity in computing must-alias information, resulting in over-approximation.

5.3 Redundant/Incorrect and Missing Annotations

Table 3 shows that our technique infers very few incorrect annotations. Figure 4 gave an example of an incorrect annotation being inferred (the `@MustCall("closeSockSync")` annotation on the class). Overall, the low rate of incorrect inferred annotations shows that our optimistic technique usually infers annotations matching the intended specification.

Table 3 shows that an average of 25% and 27% of the verifier warnings are generated because of missing annotations across C# and Java benchmarks respectively.

In C#, nearly 16% of warnings are generated because of a missing `@MustCall("")` annotation. We found that in C#, classes often implement the `System.IDisposable` interface, even though they do not manage a resource that needs to be disposed. In these cases, the `Dispose` method is used to reset the state of the instance variable and not to dispose any resources. Since we annotate the `IDisposable` type as `@MustCall("Dispose")`, by default this leads to false warnings when objects of these classes do not call `Dispose`. This issue can be addressed by annotating such classes as `@MustCall("")`, but our inference technique cannot yet infer this annotation; we leave this enhancement as future work.

For Java benchmarks, nearly 10% of warnings are generated due to re-assignment of non-final `@Owning` fields. As discussed in section 3.4.2, such cases are typically code smells. They could be addressed by adding a `@CreatesMustCallFor` annotation, but we found that exhaustive inference of these annotations was not effective. Our tool's current approach, which leads to verifier warnings any time a non-final `@Owning` field is overwritten, encourages a cleaner programming style for resources.

5.4 Run-time Performance

Table 4 gives performance results for inference and checking. For C#, we ran the inference algorithm and measured the run time as an average of 3 trials on a machine with an Intel Xeon(R) W-2145 CPU running at 3.7 GHz and 64 GB of RAM. For Java, we ran the inference algorithm and the Resource Leak Checker on a machine with a 12th Gen Intel Core i-7-12700 Processor, which has 20 cores, and 32 GB of RAM.

As noted in section 4.1, our Java implementation suffers from some inefficiencies, which are reflected in the numbers. The CodeQL-based inference implementation for C# is quite performant, always running in under 12 minutes for benchmarks with hundreds of thousands of lines of code. We believe with a more optimized architecture we could build an equally-performant inference

Table 4. Time to inferring resource management specifications. "kLoC" is non-comment, non-blank lines of code.

	kLoC	Inference time	Verification time		
			no annos	inferred annos	manually written annos
Service 1	450	2m 37s	2m 18s	3m 38s	3m 54s
Service 2	163	4m 48s	3m 31s	6m 40s	6m 44s
Service 3	147	4m 47s	3m 0s	5m 1s	5m 9s
Lucene.Net	412	11m 17s	55m 56s	58m 48s	57m 48s
EF Core	233	10m 53s	76m 27s	95m 36s	77m 48s
zookeeper	45	6m 16s	0m 33s	0m 38s	0m 28s
hadoop-hdfs	152	55m 23s	5m 48s	4m 37s	3m 22s
hbase	221	25m 48s	2m 52s	7m 59s	6m 14s

implementation for Java. Note that for C#, inference actually runs faster than running the RLC# checker. This difference is due to the RLC# verifier needing to verify properties on all paths through a method, while optimistic inference often only checks for the existence of an operation on some path. In certain cases, verification time was higher with the inferred annotations than with manual annotations; this occurred because inference discovered user-defined types that managed resources (recall that such types were not inspected during manual annotation), and the verifier was then obligated to check usages of these types for leaks.

5.5 Threats to Validity

For external validity, the primary threat for our evaluation is our choice of benchmarks. For Java, we chose benchmarks from previous work [Kellogg et al. 2021]. For C#, we strove to choose representative benchmarks. However, our inference may perform differently on other types of benchmarks or coding patterns.

Our results may also be impacted by implementation bugs, threatening internal validity. We have a suite of regression tests designed to detect such bugs, and we have also done extensive manual inspection of the output of the inference implementations on our benchmarks.

6 RELATED WORK

Static Resource Leak Detection. Several static analyses have been designed to detect resource leaks in unannotated code. Tracker [Torlak and Chandra 2010] and Grapple [Zuo et al. 2019] both employ inter-procedural dataflow analysis to detect resource leaks for Java. InferSharp [InferSharp developers 2023], built on Facebook Infer [Calcagno et al. 2015], also leverages inter-procedural analysis to detect resource leaks for C# code. Other tools use more heuristic approaches and intra-procedural analysis to detect leaks, e.g., analyses in Eclipse [Eclipse developers 2023] and PMD [PMD developers 2023].

Kellogg et al. [2021] directly compared the Resource Leak Checker to the Grapple [Zuo et al. 2019] and Eclipse [Eclipse developers 2023] tools. Compared to Grapple, the Resource Leak Checker had many fewer false negatives and ran more quickly. The Eclipse checker ran more quickly than the Resource Leak Checker, but suffered from false negatives. The key disadvantage of the Resource Leak Checker was that unlike the other two tools, the benchmarks had to be manually annotated before it could be used. The inference technique presented in this paper significantly decreases this need for manual annotation, making use of the Resource Leak Checker much easier and more compelling.

Inference. Type inference is a well-studied technique; for general background see Pierce [Pierce 2002]. Most type inference techniques either infer a set of types for a program that allow it to pass the type checker, or fail with an error message. Our scenario is quite different, as our inference must produce a useful partial solution for programs that cannot pass the type checker (due to bugs or due to code patterns that cannot be verified). Some recent techniques leverage constraints and optimization for type inference, e.g., approaches to migrate existing programs to use a gradual type system [Campora et al. 2018; Migeed and Palsberg 2020; Phipps-Costin et al. 2021]. These techniques can output a partial typing when type inference cannot fully succeed. However, the annotations required by the Resource Leak Checker are not typical type qualifiers, but instead capture ownership and aliasing protocols that reveal which code is responsible for disposing of a resource. The nature of these annotations necessitates a custom algorithm.

Vogels et al. [2011] presents an inference technique for separation logic. This approach is similar to ours in that their inference is implemented separately from their verifier, thus the inference cannot introduce unsoundness. However, there are significant differences in the inference technique: e.g., their work guesses many annotations and uses the verifier to see which are valid (like the Houdini technique [Flanagan and Leino 2001]), whereas our technique uses program analysis to discover likely specifications. Also, the properties being checked and the scale of programs being analyzed are notably different.

Hackett et al. [2006] presents a modular checker for buffer overflows that includes an inference approach similar in spirit to ours. The annotations for buffer overflow checking require a customized inference algorithm, as in our case for resource leaks. Their algorithm is also formulated using Datalog rules. And, like our technique, for practicality their algorithm is optimistic and may infer annotations that cannot be verified. We believe that this optimistic approach for inference is more general than buffer overflows and resource leak specifications, and it may be useful in other domains.

Recent work has applied machine learning to type inference [Hellendoorn et al. 2018; Peng et al. 2022; Pradel et al. 2020]. These approaches again focus on inference of traditional types for variables, and hence cannot be directly applied to inferring the kinds of annotations required by the Resource Leak Checker. Further, applying ML techniques to our inference problem could be challenging due to a lack of training data. Still, ML techniques could be complementary to our approach and help to infer better annotations in cases where our algorithm is incomplete or to tune its heuristics.

7 CONCLUSIONS

We have presented a novel technique for inference of resource management specifications, which enables broader usage of specify-and-verify tools for modular verification that no resources are leaked. Our technique leverages a custom algorithm for handling inter-related specifications of ownership, aliasing, and resource obligations. Our technique employs optimistic inference to more often capture the intended specification for code, even when the code cannot be verified. Our experimental evaluation showed that our technique to be very effective: it inferred most of the annotations developers wrote by hand, and the final true positive rate of the checker run after fully-automatic inference nearly matched the rate achieved after manual annotation.

DATA AVAILABILITY

Our inference implementation for Java will be included in a future release of the Checker Framework. Our inference tool for C# is publicly available [Gharat et al. 2023]. We have made an artifact available containing the code for both tools and scripts to run them on our open-source benchmarks [Shadab et al. 2023].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their detailed and helpful feedback. This research was supported in part by the National Science Foundation under grants 2005889, 2007024, 2312262, and 2312263, DARPA contract FA8750-20-C-0226, a gift from Oracle Labs, and a Google Research Award.

REFERENCES

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NFM 2015: 7th NASA Formal Methods Symposium*. Pasadena, CA, USA, 3–11. https://doi.org/10.4204/eptcs.188.2

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating gradual types. In *POPL 2018: Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, CA, USA. https://doi.org/10.1145/3158103

CheckerRLC 2023. Resource Leak Checker for must-call obligations. https://checkerframework.org/manual/#resource-leak-checker. Accessed 29 July 2023.

CheckerWPI 2023. Checker Framework Whole-Program Inference. https://checkerframework.org/manual/#whole-program-inference. Accessed 28 March 2023.

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *OOPSLA '98: Object-Oriented Programming Systems, Languages, and Applications*. Vancouver, BC, Canada, 48–64.

CodeQLCSharpDataflow 2023. Analyzing data flow in C#. https://codeql.github.com/docs/codeql-language-guides/analyzing-data-flow-in-csharp/. Accessed 28 March 2023.

Eclipse developers. 2023. Avoiding resource leaks. hhttps://help.eclipse.org/2023-03/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-avoiding_resource_leaks.htm. Accessed 24 March 2023.

EF Core accessed 2023. Entity Framework Core. https://github.com/dotnet/efcore#entity-framework-core.

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *FME '01: International Symposium on Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*. Berlin, Germany, 500–517.

Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*. Atlanta, GA, USA, 192–203. https://doi.org/10.1145/301618.301665

Pritam Gharat, Narges Shadab, Shrey Tiwari, Shuvendu Lahiri, and Akash Lal. 2023. Resource Leak Checker (RLC#) for C# code using CodeQL. https://github.com/microsoft/global-resource-leaks-codeql

Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. 2006. Modular checking for buffer overflows in the large. In *ICSE 2006, Proceedings of the 28th International Conference on Software Engineering*. Shanghai, China, 232–241. https://doi.org/10.1145/1134285.1134319

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *ESEC/FSE 2018: The ACM 26th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Lake Buena Vista, FL, USA, 152–162. https://doi.org/10.1145/3236024.3236051

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (May 2001), 396–450.

InferSharp developers. 2023. InferSharp. https://github.com/microsoft/infersharp/wiki/InferSharp:-A-Scalable-Code-Analytics-Tool-for-.NET. Accessed 24 March 2023.

J. B. Kam and J. D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Informatica* 7 (1977), 305–317.

Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, and Michael D. Ernst. 2023. Pluggable type inference for free. In *ASE 2023: Proceedings of the 38th Annual International Conference on Automated Software Engineering*. Luxembourg.

Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2021. Lightweight and modular resource leak verification. In *ESEC/FSE 2021: The ACM 29th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Athens, Greece, 181–192. https://doi.org/10.1145/3468264.3468576

Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2022. Accumulation analysis. In *ECOOP 2022 — Object-Oriented Programming, 33rd European Conference*. Berlin, Germany, 10:1–10:31. https://doi.org/10.4230/DARTS.8.2.22

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press. https://doc.rust-lang.org/stable/book/

Lucene.NET accessed 2023. Lucene.NET is a high performance search library for .NET. https://lucenenet.apache.org/.

Microsoft. accessed 2023. CodeQL. https://codeql.github.com.

Zeina Migeed and Jens Palsberg. 2020. What is decidable about gradual types?. In *POPL 2020: Proceedings of the 47th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New Orleans, LA, USA. https://doi.org/10.1145/3373104

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*. Seattle, WA, USA, 201–212. https://doi.org/10.1145/1390630.1390656

Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for Python. In *ICSE 2022, Proceedings of the 43rd International Conference on Software Engineering*. Pittsburgh, PA, USA, 2019–2030. https://doi.org/10.1145/3510003.3510038

Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-Based Gradual Type Migration. In *OOPSLA 2021, Object-Oriented Programming Systems, Languages, and Applications*. Chicago, IL, USA, Article 111, 27 pages. https://doi.org/10.1145/3485488

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA. https://doi.org/10.7551/mitpress/1104.003.0005

PMD developers. 2023. CloseResource. https://pmd.sourceforge.io/pmd-6.55.0/pmd_rules_java_errorprone.html#closeresource. Accessed 24 March 2023.

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: neural type prediction with search-based validation. In *ESEC/FSE 2020: The ACM 28th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Sacramento, CA, USA, 209–220. https://doi.org/10.1145/3368089.3409715

Narges Shadab, Pritam Gharat, Shrey Tiwari, Michael D. Ernst, Martin Kellogg, Shuvendu Lahiri, Akash Lal, and Manu Sridharan. 2023. Inference of Resource Management Specifications (Artifact). https://doi.org/10.5281/zenodo.8333055

Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (January 1986), 157–171.

Bjarne Stroustrup. 1994. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts.

Emina Torlak and Satish Chandra. 2010. Effective interprocedural resource leak detection. In *ICSE 2010, Proceedings of the 32nd International Conference on Software Engineering*. Cape Town, South Africa, 535–544. https://doi.org/10.1145/1806799.1806876

Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. 2011. Annotation inference for separation logic based verifiers. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*. 319–333.

Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. 2019. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *EuroSys*. Dresden, Germany, 1–17. https://doi.org/10.1145/3302424.3303972

Received 2023-04-14; accepted 2023-08-27