Unimocg: Modular Call-Graph Algorithms for Consistent Handling of Language Features

Dominik Helm
ATHENE
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
helm@cs.tu-darmstadt.de

Tobias Roth
ATHENE
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
roth@cs.tu-darmstadt.de

Sven Keidel
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany

Michael Reif
CQSE GmbH
Darmstadt, Germany
reif@cqse.eu

Mira Mezini
ATHENE
hessian.AI
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
mezini@cs.tu-darmstadt.de

ABSTRACT

Traditional call-graph construction algorithms conflate the computation of possible runtime types with the actual resolution of (virtual) calls. This tangled design impedes supporting complex language features and APIs and making systematic trade-offs between precision, soundness, and scalability. It also impedes implementation of precise downstream analyses that rely on type information.

To address the problem, we propose Unimocg, a modular architecture for call-graph construction that decouples the computation of type information from resolving calls. Due to its modular design, Unimocg can combine a wide range of different call-graph algorithms with algorithm-agnostic modules to support individual language features. Moreover, these modules operate at the same precision as the chosen call-graph algorithm with no further effort. Additionally, Unimocg allows other analyses to easily reuse type information from the call-graph construction at full precision.

We demonstrate how Unimocg enables a framework of call-graph algorithms with different precision, soundness, and scalability trade-offs from reusable modules. Unimocg currently supports ten call-graph algorithms from vastly different families, such as CHA, RTA, XTA, and *k*-*l*-CFA. These algorithms show consistent soundness without sacrificing precision or performance. We also show how an immutability analysis is improved using Unimocg.

CCS CONCEPTS

• **Software and its engineering** → **Abstraction, modeling and modularity**; **Automated static analysis**; • **Theory of computation** → *Program analysis.*

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652109

KEYWORDS

Call Graph, Type Analysis, Static Analysis, Modularization

ACM Reference Format:
Dominik Helm, Tobias Roth, Sven Keidel, Michael Reif, and Mira Mezini. 2024. Unimocg: Modular Call-Graph Algorithms for Consistent Handling of Language Features. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3652109

1 INTRODUCTION

Sound and precise call graphs are a prerequisite for inter-procedural static analysis. Over the past decades, dozens of call-graph algorithms for object-oriented programming languages have been proposed [1, 4, 9, 24, 29]. However, their implementations have inconsistent support for crucial language features, e.g., reflection, serialization, or threads—they often support these features unsoundly, or not at all [16, 17, 26].

Table 1 shows the state of affairs for the WALA [11] and Soot [30] frameworks. We generated the table by reproducing Reif et al.'s [16] study of soundness[1] of JVM-language call-graph algorithms. We used the current version of their benchmark[2]—a suite of manually annotated tests—and current versions of WALA (1.5.7) and Soot (4.4.1) for the call-graph algorithms *Class-Hierarchy Analysis* (CHA) [4], *Rapid-Type Analysis* (RTA) [1], *Control-Flow Analysis* (CFA) [24], and Soot's default configuration of SPARK [13].

While the exact numbers have changed, the overall picture stays the same as reported by Reif et al. [16]: Language feature support varies significantly not only across frameworks but even across algorithms within the same framework. In general, more precise call-graph algorithms become less sound. All call-graph algorithms of WALA and Soot fail to soundly analyze about half of the test cases. For some features, call-graph algorithms even fail all test

[1] None of these call-graph algorithms are sound in the mathematical sense. In this paper, we refer to the term *soundness* as a lower number of missing edges in call graphs. This is in line with related work [16, 26].
[2] https://github.com/opalj/JCG

Table 1: Soundness of call-graphs for different JVM features

	WALA			Soot		
Feature	**CHA**	**RTA**	**0-CFA**	**CHA**	**RTA**	**SPARK**
Non-virtual Calls	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6
Virtual Calls	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4
Types	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6
Static Initializer	◐ 4/8	◐ 7/8	◐ 6/8	◐ 7/8	◐ 7/8	◐ 7/8
Java 8 Interfaces	● 7/7	● 7/7	● 7/7	● 7/7	● 7/7	● 7/7
Unsafe	● 7/7	● 7/7	○ 0/7	● 7/7	● 7/7	○ 0/7
Class.forName	◐ 2/4	● 4/4	● 4/4	◐ 2/4	◐ 2/4	◐ 2/4
Sign. Polymorph.	○ 0/7	○ 0/7	○ 0/7	○ 0/7	○ 0/7	○ 0/7
Java 9+	● 2/2	◐ 1/2	◐ 1/2	● 2/2	● 2/2	● 2/2
Non-Java	● 2/2	● 2/2	● 2/2	○ 0/2	○ 0/2	○ 0/2
MethodHandle	◐ 2/9	◐ 2/9	○ 0/9	◐ 2/9	◐ 2/9	○ 0/9
Invokedynamic	○ 0/16	◐ 10/16	◐ 10/16	◐ 11/16	◐ 11/16	◐ 11/16
Reflection	◐ 2/16	◐ 3/16	◐ 6/16	◐ 2/16	◐ 2/16	○ 0/16
JVM Calls	◐ 2/5	◐ 3/5	◐ 3/5	◐ 4/5	◐ 4/5	◐ 3/5
Serialization	◐ 3/14	◐ 1/14	◐ 1/14	◐ 3/14	◐ 1/14	◐ 1/14
Library Analysis	◐ 2/5	◐ 2/5	◐ 1/5	◐ 2/5	◐ 2/5	◐ 2/5
Class Loading	○ 0/4	○ 0/4	○ 0/4	○ 0/4	○ 0/4	○ 0/4
DynamicProxy	○ 0/1	○ 0/1	○ 0/1	○ 0/1	○ 0/1	○ 0/1
Sum (out of 123)	51 (41%)	65 (53%)	57 (46%)	65 (53%)	63 (51%)	51 (41%)

Algorithms within each framework are ordered by increasing precision
Soundness: all ●, some ◐, or no ○ test cases passed soundly

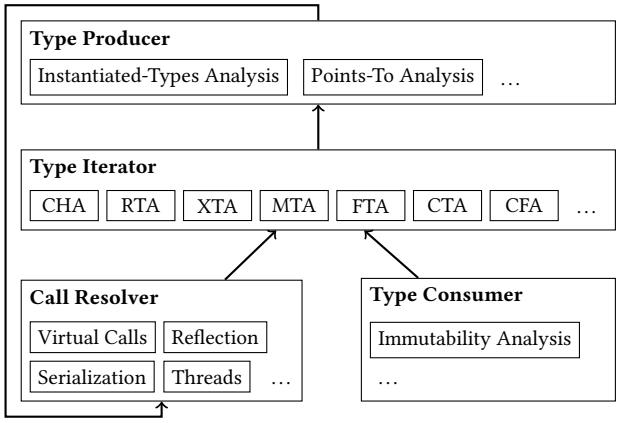

Figure 1: Unimocg's modular call-graph architecture

cases, an indication that they may not support the feature explicitly. In some cases, less precise call-graph algorithms like CHA and RTA pass tests due to excessive imprecision rather than to actual feature support. Surprisingly, WALA's CHA is not only less precise but apparently also less sound than WALA's RTA. The inconsistent support of language features makes it difficult for users to systematically choose an appropriate call-graph algorithm. One has to know in detail which language features each algorithm supports soundly and cannot decide based on precision and performance alone. This also makes it difficult for researchers to compare call-graph algorithms and implementations in terms of precision and performance.

In this paper, we analyze reasons for this observed inconsistency and propose a solution to the problem. In short, the problem is that different call-graph algorithms handle language features in specific ways by making specific use of different kinds of information they have access to. For example, a CFA algorithm can soundly handle more reflection calls because it has access to pointer information. CHA and RTA handle reflection differently because they do not have access to pointer information. As a result, code that handles call resolution for individual language features is coupled to specific call-graph algorithms, which makes it difficult to reuse that code across different call-graph algorithms. Developers wanting to support new language features have to duplicate code to support dissimilar algorithms. Thus, the available resources and priorities of developers of a certain framework determine which features are supported by which algorithm; maintenance is also complicated as features evolve.

The work presented in this paper demonstrates that it is possible to implement a variety of call-graph algorithms with consistent handling of language features. Specifically, we propose Unimocg (*UNIfied MOdular Call Graphs*), a novel architecture for modular implementation of call-graph algorithms that decouples the implementation of the following concerns: (1) computation of type information, (2) interpretation of type information, (3) resolution of calls, and (4) analyses that depend on type information. Fig. 1 overviews the components that handle these concerns in Unimocg and their relations.

Type producers compute information about the runtime types of variables and fields. *Type iterators* interpret this type information and make it available to call resolvers and type consumers—keeping them decoupled from type producers. *Call resolvers* resolve method calls or calls of language features such as reflection, serialization, or threads. They query a type iterator for type information about call receivers or arguments of reflective calls; in turn, the information about resolved method calls is used by type producers. *Type consumers* are static analyses that depend on type information but do not contribute to call-graph construction. Without type iterators, type consumers would have to rely on imprecise type information, e.g., provided by static types, to avoid dependence on a specific call-graph algorithm. For instance, the immutability analysis by Roth et al. [20] uses imprecise type information from static types and the class hierarchy.

Unimocg's modular architecture enables deriving call graphs with consistent coverage of language features and hence soundness by reusing and combining type producers, type iterator, and call resolvers in a plug-n-play manner. We show that Unimocg enables a wide range of call-graph algorithms to share the same support for language features such as reflection and serialization, thus ensuring consistent soundness, by implementing ten algorithms from different families: CHA [4], RTA [1], the XTA family with MTA, FTA, and CTA [29], as well as *k-l*-CFA-based algorithms 0-CFA, 0-1-CFA, 1-0-CFA, and 1-1-CFA [9].

Type consumers also benefit from the modular architecture by reusing precise type information computed for call-graph construction. To showcase this, we reimplemented the immutability analysis by Roth et al. [20] as a type consumer in Unimocg. Unimocg's separation of call-graph construction from the computation of type information ensures consistent and improved precision of the immutability analysis when combined with more precise call graphs.

Unimocg benefits both users and developers of static analyses (call-graph and other analyses). Users can rely on consistent soundness and can systematically choose appropriate algorithms for their

respective applications considering only their intuition about the relative precision and performance of different algorithms. Analysis developers, on the other hand, can easily extend Unimocg to support new language features across all available algorithms or add new call-graph and/or other analysis algorithms while retaining all available feature support.

As our evaluation shows, Unimocg's benefits do not come at the cost of precision or performance compared to state-of-the-art static analysis frameworks.

In summary, we make the following contributions:

- We analyze the reasons for inconsistent soundness in state-of-the-art call-graph algorithms (Section 2)
- We propose Unimocg, a novel architecture for implementing call-graph analyses modularly (Section 3). Unimocg allows deriving various call graphs with consistent soundness. In particular, we show how resolution of features like reflection can be decoupled from the call-graph algorithm.
- We discuss how Unimocg benefits analyses that depend on type information but do not participate in call-graph construction (Section 3.5).
- We use Unimocg to implement families of different call graphs and measure soundness, precision, and performance compared to the state of the art and evaluate their impact on a state-of-the-art immutability analysis (Section 4).

The current implementation of Unimocg focuses on call-graph construction for JVM languages, but the addressed problems and the Unimocg architecture are language agnostic.

2 PROBLEM STATEMENT

We analyze two problems with existing call-graph algorithms responsible for the observed soundness inconsistency (Table 1).

Problem 1: Coupling of Call Resolution of Language Features to Base Call-Graph Algorithms. Modern programming languages have many features, which are difficult to analyze. Three such Java features are reflection, (de)serialization, and threads. *Reflection* [12] dynamically instantiates classes and calls methods based on runtime strings and types. To resolve reflective calls, a call-graph analysis needs to statically determine strings for the class and method names. It also has to determine receiver and class objects as well as argument types. *(De)serialization* [22] writes or reads Java objects from a stream of bytes, e.g., a file. For (de)serialization, the JVM invokes special methods, e.g., `readResolve`, that must be handled by the call-graph analysis. To resolve a call on a deserialized object, a call-graph analysis also needs to determine the types of objects in the byte stream. *Threads* are started by calling the built-in `Thread.start` method, which leads to the JVM invoking `Thread.run`; hence, it requires specific handling by call-graph algorithms. What complicates the problem further is that these features can be used in combination, e.g., threads may start `Runnable` objects loaded via reflection.

Advanced features induce unique challenges for different call-graph algorithms—hence, each call-graph analysis typically treats them specifically. For example, reflection is easier to handle by call-graph algorithms that have allocation information and deserialization is easier for imprecise algorithms that over-approximate the possible classes deserialized.

Handling language features differently creates coupling between call resolution of features and the base call-graph algorithms and violates separation of concerns. For example, WALA's CHA does not use the call-graph-builder facilities of RTA and CFA but its own, redundant implementation of some features. Thus, WALA's RTA algorithm handles static initializers differently from WALA's CHA—with RTA, they should be deemed reachable only for classes actually instantiated. In Soot, the call-resolution code for RTA and Spark is strongly coupled to the call-graph builder class, which not only resolves standard calls, but is also responsible for recognizing reflection. Supporting new language features would thus require changing the call-graph builder class. Both Soot and Wala resolve only explicit calls. Features that are not explicit calls such as static initializers do not use the same call-resolution facilities.

One might argue that this is only the result of missing care in the implementation and that more implementation effort could fix these inconsistencies. While true, without shared language feature support, this places a lot of burden onto developers of the call-graph analyses, both in terms of care and implementation effort to maintain feature support individually for different algorithms, and feature support can again diverge over time.

Problem 2: Different Type Information. Different call-graph algorithms require different type information to resolve virtual calls. For example, CHA requires only the declared types, RTA additionally requires information about the classes that are instantiated anywhere in the program, and CFA requires the precise information produced by a pointer analysis.

These different representations typically require calls to be resolved specifically for each call-graph algorithm. An implementation for call resolution for CHA is not compatible with RTA and vice-versa because the code to retrieve subtypes from the class hierarchy is different from code to retrieve suitable types from a global type set. This is especially true if the global type set for RTA is constructed on the fly during call-graph construction, i.e., it constantly changes, while type-hierarchy information is constant. The same problem holds for CHA and CFA or RTA and CFA—in fact, for any two call-graph algorithms.

A potential solution is to adopt a single representation for type information for all algorithms—typically, a points-to representation. For instance, Soot and WALA do not directly implement RTA, but emulate it by means of a points-to analysis. This strategy is, however, inefficient for algorithms that do not require such an intricate representation. Our validation (Section 4) of RTA in both Soot and WALA confirms this. Using a single representation also closely couples the resolution of calls to the computation of type information, exchanging the single representation for a different one, or one using a different algorithm is not easily done in WALA or Soot. The declarative framework Doop [3] also uses a single representation, computing points-to information using Datalog queries. Unlike WALA or Soot, Doop only supports algorithms of the CFA family, and it is unclear whether or how efficiently supporting algorithms like CHA or RTA could be implemented in Doop while reusing its call-resolution Datalog queries that use points-to information.

Summary of Problems. Together, the outlined problems make it difficult to support language features across multiple call-graph algorithms, thus complicating call-graph implementation. Complex

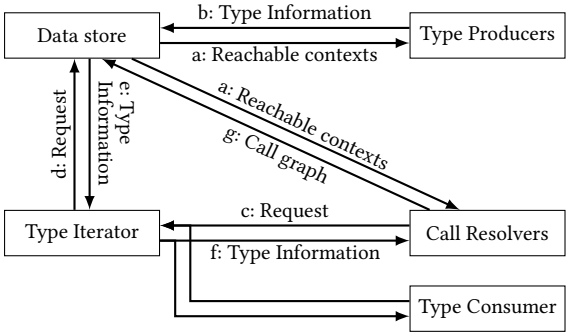

Figure 2: Interaction of components

code for resolving language features needs to be re-implemented over and over for different base algorithms, and different kinds of type information need to be handled in implementing resolution code. Ultimately this leads to soundness inconsistencies (Table 1). For instance, soundness with regard to static initializers differs between WALA's CHA and RTA in an unexpected way. WALA's RTA algorithm is in many cases more sound than WALA's CHA. This is surprising because the less precise CHA should theoretically be more sound.

Our Solution in a Nutshell. To address the problems, we decouple the call resolution of special language features from the base call-graph algorithm and capture them in independent *call-resolver* modules. To enable this decoupling, we introduce the *type iterator*, an abstraction layer that retrieves and interprets the different type representations produced by different base algorithms (*type producers*). This way, call resolvers for individual language features can be implemented only once, by using the type iterator as a unified interface to access type information. Thus, they use the most precise available type information, while being independent of its internal representation or computation. Individual call resolvers are decoupled from each other and agnostic of the call-graph algorithm. They can be plugged into call-graph algorithms without changes to existing code.

Our approach addresses problem 1 by having the decoupled call resolvers collaborate to resolve calls for different language features. It addresses problem 2 by having type information be kept in the most efficient representation for each individual base algorithm. We show that this approach leads to more consistent soundness. Furthermore, it improves the maintainability of call-graph algorithms, as one can easily add, reuse, or exchange call resolvers to tune precision and performance.

3 UNIMOCG MODULAR ARCHITECTURE

We start with an overview of Unimocg's components. We then describe individual components in detail and discuss how they collaborate despite being decoupled.

3.1 Architectural Overview

Unimocg consists of four types of components (Fig. 1): *Type producers* analyze the code to compute the possible runtime types of local variables and fields. A *type iterator* provides a unified view on this information for other components to use. *Call resolvers* use type information through the type iterator to resolve method calls that result from different language features. Finally, *type consumers* are further analyses that use type information but do not resolve calls.

Components are decoupled from each other using interfaces and communicate indirectly via a central data store; a fixed-point solver integrated therein serves as an intermediary. To bootstrap the process, the store is initialized with a set of entry-point contexts[3], e.g., the analyzed program's main method(s).

Fig. 2 depicts how components interact: Whenever a new reachable context is discovered, the solver triggers type producers and call resolvers (a). Type producers process the new context and return new type information to the data store (b). Call resolvers analyze the new context and request data from the type iterator (c); the latter forwards the request to the store (d), which returns to the type iterator whatever type information is currently available (e). The type iterator interprets the information and forwards the result to call resolvers (f). Also, if step (b) found additional information, the data store notifies the type iterator to forward this information to call resolvers and type consumers that requested it earlier (e & f). Finally, call resolvers add new edges to the call graph (g), which may reveal more reachable contexts and the cycle repeats. When no new edges or type information are found anymore, the analysis reached a fixed point and terminates. Note that edges are never changed or removed from the store, thus termination is guaranteed.

3.2 Type Producers

Type producers compute type information required by other components. For instance, a type producer for an RTA call graph calculates which classes the program instantiates, while a type producer for a CFA call graph computes points-to information of local variables. A call graph may also use multiple type producers, e.g., we may split the points-to type producer for CFA into multiple modules, each handling different language features, e.g., `java.lang.System.arraycopy`. A call graph may not need any type producer, e.g, a CHA algorithm can compute type information directly from the class hierarchy without a dedicated type producer.

Type producers represent type information in an algorithm-specific way. For example, an RTA type producer represents its type information as a global type set, while different CFA type producers for different language features represent their information as points-to sets and set union is used to combine their results. Some algorithms like k-l-CFA with $l \geq 1$ additionally provide allocation data, while other type producers cannot provide such information. Allocation data may be needed by specific call resolvers and type consumers, e.g., resolving a reflective call of `Method.invoke` requires knowledge about the particular `Method` object involved.

Despite employing algorithm-specific representations, type producers implement a common interface (shared with call resolvers). Listing 1 shows pseudocode for this interface. Global singletons are used to retrieve `dataStore` and `typeIterator`, the actual analysis is defined in method `analyze`. Different type producers implement `analyze` in specific ways. It is executed once for each context `ctx` found reachable and computes the respective type information.

[3] A context is the (context-sensitive) abstraction of a method invocation [14]

```
1 interface CallGraphAnalysisModule:
2     dataStore: DataStore := [...]
3     typeIterator: TypeIterator := [...]
4     fun analyze(context: Context)
```

Listing 1: Interface for Type Producers & Call Resolvers

```
1 [...]
2 for statement in method.statements:
3     if statement is Assigment(local, call: Call):
4         callTargets := dataStore.get((context, call), CallTargets)
5         for target in callTargets if target is constructor:
6             newObject := PointsTo(context,call.programCounter,
                target.class)
7             dataStore.add((context, local), PointsTo, newObject)
8 [...]
```

Listing 2: Points-To Type Producer (Excerpt showing points-to data creation on constructor invocations)

Type producers are agnostic of how calls are resolved. They are triggered by the store for all reachable contexts, regardless of how the latter are computed. We illustrate this in Listing 2: It shows an excerpt of the points-to type producer's `analyze` method where points-to objects are created whenever the analysis of the current context method finds a call whose target is a constructor. The type producer uses only the information in the `callTargets` that are retrieved from the data store, the implementation is agnostic of the call resolver that found the target. In particular, the constructor invocation could be the result of reflection (e.g., `Class.newInstance`) or of deserialization instead of a direct call.

3.3 Type Iterator

Type iterators implement the iterator pattern [8] to allow retrieving and iterating over information on the possible runtime types of a local variable or a field from the data store in a uniform way. Type iterators and type producers go hand in hand (e.g., the RTA type iterator requires an RTA instantiated-types analysis to be executed), except for CHA where type iteration happens on the fly.

Despite their close relation, we separate type producers from type iterators for two reasons: First, one can have different type iterators provide different views on the type information of a single producer. For example, we can have different iterators for a single points-to type producer, each for different context sensitivities. Second, a single type iterator can provide an aggregated view over the information produced by multiple type producers. For example, a single type iterator for some given context sensitivity can aggregate the information of several points-to type producers—a basic one for local variables, etc. and additional ones for advanced features, e.g., native methods or reflection.

We abstract over specific type iterators with a unified interface `TypeIterator`. Listing 3 shows the methods that operate on local variables; analogous methods that operate on fields are omitted for brevity. The generic type `Context` specifies the type of context used, e.g., call strings. Methods `foreachType` and `foreachAlloc`

```
1 interface TypeIterator[Context]:
2   fun foreachType(var: Local, context: Context,
      handleType: Type -> ())
3   fun foreachAlloc(var: Local, context: Context,
      handleAlloc: (Type, Context, ProgramCounter) -> ())
4
5   fun newContext(method): Context
6   fun expandContext(old: Context, callee: Method): Context
7   [...]
```

Listing 3: Type Iterator Interface

iterate over types and allocations for a local variable `var` in a certain context `ctx`. The interface also defines two methods for iterating on incremental updates of the type data, which we omit for brevity. `newContext` returns a new context based on a method and `expandContext` extends an existing context `old` as necessary for context-sensitive analyses like *k*-*l*-CFA. This enables type producers to support different types of context. Call resolvers on the other hand are oblivious to the type of context and treat it as a method.

Unimocg includes type iterators for CHA, RTA, XTA, MTA, FTA, CTA, 0-CFA, 1-0-CFA, 0-1-CFA, and 1-1-CFA. In the following, we discuss three exemplary iterator instances, shown in Listing 4, for CHA, RTA, and CFA, to show how they retrieve type information from the data store and how they make it available to call resolvers and type consumers. We do not discuss the remaining iterators, but Unimocg is available under an open-source BSD 2-clause license as part of the OPAL framework.[4]

The CHA type iterator does not need data from the store as the class hierarchy is computed a priori. Hence, method `foreachType` simply iterates over all subtypes of the variable's declared type. The RTA iterator resolves the variable's types based on which types may be instantiated. It retrieves the global set of instantiated types from the central data store, then filters this set to only the subtypes of the variable's declared type; finally, it iterates over these types. The *k*-CFA type iterator resolves the variable's types based on contextual points-to information using a *k*-truncated call context [23] (line 29). Given such a call context, the methods `foreachType` and `foreachAlloc` retrieve the set of allocation sites from a context-sensitive points-to analysis from the data store and iterate over only the respective types, respectively all allocation sites.

3.4 Call Resolvers

Call resolvers use information obtained from type iterators to resolve call sites to possible target contexts. Like *type producers*, they implement the interface in Listing 1. For illustration, we discuss two call resolvers of different complexity.

Listing 5 shows how the call resolver for regular calls uses the type iterator to resolve virtual method calls: After the `analyze` method found a virtual method call `call`, it iterates over all possible runtime types of the receiver object (Line 3). Once the types are known, resolving the call to a callee method (Line 4), creating a target context for the callee (Line 5) and the call edge (Line 6), and adding it to the call graph (Line 7) are standard steps in all call-resolution code.

[4] https://www.opal-project.de

```
1  class CHATypeIterator extends TypeIterator[MethodContext]:
2    fun foreachType(local, context, handleType):
3      for t in var.declaredType.subtypes:
4        handleType(t)
5    [...]
6
7  class RTATypeIterator extends TypeIterator[MethodContext]:
8    fun foreachType(local, context, handleType):
9      for t in dataStore.get(InstantiatedTypes):
10       if t is subtype of local.declaredType:
11         handleType(t)
12   [...]
13
14 class CFATypeIterator(k: Int) extends TypeIterator[Callstring]:
15   fun foreachType(local, context, handleType):
16     allocations := dataStore.get((context, local), PointsTo)
17     for a in allocations:
18       handleType(a.type)
19
20   fun foreachAlloc(local, context, handleAlloc):
21     allocations := dataStore.get((context, local), PointsTo)
22     for a in allocations:
23       handleAlloc(a.type, a.context, a.programCounter)
24
25   fun newContext(method):
26     List(method)
27
28   fun expandContext(old, callee):
29     old.take(k).prepend(callee)
30   [...]
```

Listing 4: Type Iterators

```
1 [...]
2 if instruction is virtual call call with receiver variable r:
3   typeIterator.foreachType(r, context, receiverType -> {
4     callee := resolveCall(call, receiverType)
5     target := typeIterator.expandContext(context, callee)
6     callEdge := CallEdge(context, call.programCounter, target)
7     dataStore.add((context, call), CallTargets, callEdge)
8   })
9 [...]
```

Listing 5: Basic Call Resolver (Excerpt showing resolution of virtual calls)

In Listing 6, we show an excerpt of the reflection call resolver's `analyze` method. It is more complex, but also directly uses the type iterator: When the reflection resolver finds a call, it checks whether this is a `Method.invoke` call (Lines 2-3). It takes this information from the global store, no matter which call resolver found that call edge. For calls to `Method.invoke`, the resolver gathers information about the receiver and parameters of the reflectively invoked method (Lines 4-5); this step (method `getObjects`, which is not

```
1 [...]
2 targets := dataStore.get((context, call), CallTargets)
3 if ∃t ∈ targets : t.class=Method ∧ t.name="invoke":
4   receivers := getObjects(t.params.first)
5   params := t.params.tail.map(getObjects)
6   method := t.receiver
7   typeIterator.foreachAlloc(method, context, alloc -> {
8     newTargets := findTargets(alloc, receivers, params)
9     for target in newTargets:
10      callEdge := CallEdge(context, call.programCounter,
          target, receivers, params)
11      dataStore.add((ctx, call), CallTargets, callEdge)
12  })
13 [...]
```

Listing 6: Reflection Call Resolver (Excerpt showing resolution of `Method.invoke`)

shown here) uses the type iterator's `foreachAlloc`. The reflection resolver then iterates over the possible `Method` objects, as they encode which method can be invoked (Line 7). As in regular virtual-call resolution, the final steps are finding possible target methods (Line 8) and adding a corresponding call edge to the store (Line 11).[5] As Line 7 shows, allocation data is used to resolve reflection. Where type producers do not provide such data, like for CHA or RTA, the type iterator instead iterates over intra-procedural allocation sites from def-use information and signifies if this is incomplete.

Call resolvers are decoupled from each other (and from type producers) via the data store. Yet, they collaboratively compute the call graph. The information contained in the call edge in Line 10 on the receiver and the parameters is made available to other call resolvers and type producers through the store. E.g., in Lines 4-6, we get this data from the individual `target` call edge, not from the `call` in the analyzed code. This is important to allow resolution of chained indirect invocations. For instance, if the `Method` object represented `Method.invoke` and `receivers` contained further `Method` objects, this chained invocation still can be resolved by the code in Listing 6.

Multiple call resolvers that cover different language features, e.g., virtual calls, reflection, threads, or serialization, collaboratively construct the call graph. By combining a set of call resolvers, one can configure the soundness of a call graph. Individual resolvers are reusable across different algorithms because they only depend on the common interface of type iterators. As a result, it is easy to ensure consistent feature handling across different algorithms.

3.5 Type Consumers

Type information is useful for a range of analyses beyond call-graph construction. For instance, to determine the immutability of a field `f`, an immutability analysis may use the types of objects that `f` could refer to [20]. We model such analyses as so-called *type consumers*. They access type information through the type iterator interface, which decouples them from the call-graph algorithm that produces

[5] Finding target methods uses a simple analysis of constant strings aided by Unimocg providing access to allocation sites. A more sophisticated string analysis can be implemented as a type consumer and used instead for improved soundness and precision.

this information. This allows to easily change the call-graph algorithm without modifying the type-consumer analyses; by doing so, we can fine-tune the precision and performance of type consumers. As such, they are conceptually the same as call resolvers, but they do not (directly) participate in call-graph construction, so we discuss them separately. As type consumers depend on the *single* type iterator, they consider type information with exactly the precision of the chosen iterator. This ensures that all type consumers operate on a consistent level of soundness, precision, and performance. In contrast, different parts of a monolithic analysis may use varying levels of hard-coded precision, hindering a systematic exploration of precision, soundness, and performance trade-offs.

4 VALIDATION

We implemented the architecture presented in the previous section in the OPAL framework [6, 10]. Our implementation is available open source under a BSD 2-clause license.[6] OPAL's blackboard architecture with its fixed-point solver [10] serves as the central data store of Unimocg and enables analysis modules to collaborate while being fully decoupled. The Unimocg architecture is, however, framework-independent and can be instantiated on top of any interaction infrastructure that supports decoupled components to collaborate. We use the OPAL-based implementation to empirically validate our claims by examining the following research questions:

- **RQ1** Does Unimocg enable deriving families of call graphs from reusable modules?
- **RQ2** Do the resulting call graphs exhibit consistent soundness?
- **RQ3** What is the impact of Unimocg's modular architecture and increased soundness on precision and performance?
- **RQ4** What is the impact of Unimocg on type consumers?

All measurements were performed in a Docker container that is provided as an artifact.

4.1 Deriving Call-Graph Families

To answer **RQ1**, we implemented various type producers, type iterators, and call resolvers and used them to derive ten different call-graph algorithms.

The implemented *type producers* are: (i) a global instantiated-types analysis, (ii) a parameterized propagation-based instantiated-types analysis, (iii) a parameterized points-to analysis augmented by type producers for Java APIs that require special handling, e.g., `java.lang.System.arraycopy` and `sun.misc.Unsafe`.

The implemented *type iterators* are: (i) CHA and RTA iterators from Listing 4, (ii) an iterator that is parameterized to support the propagation-based algorithms XTA, MTA, FTA, and CTA [29], (iii) traits for different context sensitivities and points-to set representations of *k*-*l*-CFA call graphs.

The implemented *call resolvers* are: (i) a virtual and non-virtual call resolver, (ii) call resolvers for reflection, serialization, threads, static initializers, finalizers, the `doPrivileged` API provided by the class `java.security.AccessController`, and a number of important native methods from the JDK's class library, (iii) an alternative call-resolver for reflection that uses information from the dynamic analysis *Tamiflex* [2].

[6] https://www.opal-project.de

Combining these type producers, type iterators, and call resolvers, we derived ten different call-graph algorithms: CHA, RTA, XTA, MTA, FTA, CTA, 0-CFA, 0-1-CFA, 1-0-CFA, and 1-1-CFA. For CHA, we used the CHA iterator without a type producer, as it solely depends on the precomputed class-hierarchy type information. To derive RTA, we combined the RTA iterator with the global instantiated-types type producer. For XTA, MTA, FTA, and CTA, we combined the respective iterators with the propagation-based instantiated-types type producer. Lastly, to derive the CFA variants, we combined the respective CFA type-iterator traits with the points-to type producers. Importantly, we could reuse the same call resolvers across all algorithms although the type information produced differs and call resolvers need different information.

Due to the abstraction that is the common type iterator interface, we can easily derive a family of call-graph algorithms with varying levels of precision and performance. By selecting an appropriate type iterator and corresponding type producer(s), users select the precision of type information to be computed, i.e., the precision of the call graph. The selection of call resolvers for language features is orthogonal to the selection of type iterators and type producers. This makes it easy for analysis developers to add new algorithms (as a combination of type iterator and type producer) or call resolvers for new features and reuse all existing components with them.

- *Unimocg enables deriving families of different call graphs by modularly composing individual components.*
- *All call resolvers are reusable across all algorithms.*

4.2 Soundness Consistency

To answer **RQ2**, we executed the benchmark of Reif et al. [16] on five of Unimocg's algorithms—CHA, RTA, XTA, 0-CFA, 1-1-CFA—representative for the different families of algorithms; other algorithms from these families (e.g., MTA instead of XTA) show similar results. Reif et al.'s benchmark measures missing edges (false negatives) caused by insufficient support of individual language features. We used this instead of measuring recall on real-world applications, as there is no suitable ground truth for recall on real-world applications. Also, this would only provide a global view on false negatives, i.e., the overall number of missing call-graph edges independent of language features; thus, we would not be able to answer if Unimocg improves consistency of language feature support.

Table 2 shows the results: Unimocg's call-graph algorithms exhibit high and consistent language feature support. They soundly pass between 79% and 81% of all test cases. In contrast, WALA and Soot pass between 41% and 53% of test cases (Table 1). Consistent feature support in Unimocg is due to its call-graph algorithms sharing the same call resolvers.

Whereas in Soot and WALA, there is a difference in soundness of 12 percentage points (pp), Unimocg shows now such differences. Except for 1-1-CFA, its algorithms show the identical soundness profile and are fully consistent with each other. The 2 pp difference for 1-1-CFA is easily explained: 1-1-CFA has access to interprocedural allocation site information, which allows it to more soundly and precisely resolve reflection. Remaining unsoundness in Unimocg is the result of not yet implemented call resolvers. Currently, we still lack resolvers for class loading and dynamic proxies, complex features not supported by WALA or Soot either. Once respective

Table 2: Soundness of Unimocg's call-graph algorithms

Feature	CHA	RTA	XTA	0-CFA	1-1-CFA
Non-virtual Calls	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6
Virtual Calls	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4
Types	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6
Static Initializer	● 8/8	● 8/8	● 8/8	● 8/8	● 8/8
Java 8 Interfaces	● 7/7	● 7/7	● 7/7	● 7/7	● 7/7
Unsafe	● 7/7	● 7/7	● 7/7	● 7/7	● 7/7
Class.forName	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4
Sign. Polymorph.	● 7/7	● 7/7	● 7/7	● 7/7	● 7/7
Java 9+	● 2/2	● 2/2	● 2/2	● 2/2	● 2/2
Non-Java	● 2/2	● 2/2	● 2/2	● 2/2	● 2/2
MethodHandle	● 9/9	● 9/9	● 9/9	● 9/9	● 9/9
Invokedynamic	◐ 11/16	◐ 11/16	◐ 11/16	◐ 11/16	◐ 11/16
Reflection	◐ 10/16	◐ 10/16	◐ 10/16	◐ 10/16	◐ 13/16
JVM Calls	◐ 3/5	◐ 3/5	◐ 3/5	◐ 3/5	◐ 3/5
Serialization	◐ 9/14	◐ 9/14	◐ 9/14	◐ 9/14	◐ 9/14
Library Analysis	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5
Class Loading	○ 0/4	○ 0/4	○ 0/4	○ 0/4	○ 0/4
DynamicProxy	○ 0/1	○ 0/1	○ 0/1	○ 0/1	○ 0/1
Sum (out of 123)	97 (79%)	97 (79%)	97 (79%)	97 (79%)	100 (81%)

Algorithms are ordered by increasing precision
Soundness: all ●, some ◐, or no ○ test cases passed soundly

call resolvers are implemented, they can be used consistently for all algorithms. To sum up, the sources of unsoundness in Unimocg are explainable and resolving them only requires further engineering effort, but no changes to existing call resolvers.

▪ *Unimocg shows consistently high soundness compared to other frameworks. This is the result of reusing the same call-resolver modules across all call-graph algorithms.*

4.3 Impact on Precision and Performance

Looking only at test cases passed could lead to incorrect conclusions if test cases were passed only due to excessive imprecision or come at a significant performance cost. Thus, we show that Unimocg's consistent soundness does not compromise precision or performance (**RQ3**). To this end, we compare three different call-graph algorithms: CHA, RTA, and 0-CFA. These algorithms are available in all frameworks being compared, Unimocg, Soot, and WALA (0-CFA only for WALA and Unimocg). We run the subject algorithms on five Java applications from XCorpus [5], which were also used by Reif et al. [16]. We used the Adoptium OpenJDK 1.8.0_342-b07 that worked with all frameworks. The different frameworks by default exclude different parts of the JDK from the analysis, thus, we configured all frameworks to analyze the full JDK for comparability. Soot's RTA, however, can not analyze the full JDK 8 as that contains MethodHandle constants that Soot's RTA is not prepared to handle (Soot's SPARK has the same problem). We thus ran Soot RTA with its default configuration excluding large parts of the JDK. The results are shown in Table 3.

Precision. Following common methodology [25, 29], we measure precision by counting the number of reachable methods—more precise call graphs have fewer reachable methods.

Unimocg's call-graph algorithms do not suffer precision degradation compared to Soot's and WALA's. Unimocg's CHA is comparable to both Soot's and WALA's (~5% and ~18% more methods, respectively). Soot's RTA, even though excluding large parts of the JDK, has on average 1.7x and WALA's has 7x the reachable methods of Unimocg's RTA. WALA's CFA has on average 2x the reachable methods of Unimocg's. While precision across different frameworks differs significantly and is difficult to compare precisely, the numbers clearly indicate that Unimocg's algorithms do not suffer from systematic imprecision, on the contrary. This is explainable: modularity helps not only to cover more features orthogonally, increasing soundness; it also makes it easier to implement more precise call resolution because call resolvers make use of the precise type information provided by type producers.

Also note that for Unimocg, the number of reachable methods decreases significantly when we use more precise type producers; e.g., its RTA identifies around 90% fewer reachable methods than its CHA. This matches the expectations of call-graph users and shows that reusing the same call-resolver modules across call-graph algorithms does not impair their relative precision. Specifically, individual call resolvers use the type information gathered by the chosen type producer and thus work at a consistent level of precision. This is not possible if the computation of type information is tightly coupled to the resolution of virtual calls; in this case, modules for other language features would rely on a fixed, potentially ad-hoc, method of computing type information.

Performance. Table 3 reports analysis runtime in seconds as median of 3 executions on a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM.

Unimocg's call-graph algorithms do not suffer performance degradation, either. The most notable finding is the difference in RTA performance. Unimocg's RTA is on average 14x and 35x faster than RTA of Soot (again, excluding large parts of the JDK), resp. WALA. This is noteworthy because to enable reusing call-graph modules, e.g., for the one resolving reflection, Soot and WALA emulate RTA by a points-to analysis. Our evaluation indicates that this approach seems to come at a significant performance cost. Unimocg's 0-CFA is 2.4x slower than WALA's 0-CFA across applications in the benchmark except *jext*, on which WALA's 0-CFA took over 36 minutes and Unimocg only took 86 seconds. Unimocg's CHA is 5x slower than WALA's, but 6x faster than Soot's. We attribute this to OPAL's intermediate representation, which employs abstract interpretation [18] to provide refined type information but needs more computation time per reachable method; Soot's Jimple [31], while not based on abstract interpretation, offers similar information. Note that performance differences between frameworks may be due to implementation details beyond our proposed architecture. Thus, this experiment cannot be used to draw conclusions about absolute performance, only show that Unimocg does not compromise performance because of its modularity.

We conclude that Unimocg's modular architecture does not compromise performance. This is explainable: the main indirection we add to OPAL is cheap—two method calls on the type-iterator object.

▪ *Unimocg's modular architecture does not compromise on precision or performance; this is indicated by the comparison to state-of-the-art frameworks and consistent relative precision across algorithms.*
▪ *Crucially, Unimocg enables reuse of modules across different algorithms without relying on inefficient emulation; this benefit is indicated by Unimocg significantly outperforming Soot and WALA's RTA implementations, which are emulated by points-to algorithms.*

Table 3: Precision and Performance of different Call-Graph Algorithms

	CHA						RTA						0-CFA			
	Soot		WALA		Unimocg		Soot[7]		WALA		Unimocg		WALA		Unimocg	
Project	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time
jasml	125 408	247	111 761	8	131 680	39	17 258	105	98 497	563	10 919	16	16 100	16	9 178	38
javacc	126 230	236	112 582	8	132 508	36	18 052	118	99 322	541	11 711	15	16 884	17	9 970	46
jext	127 960	245	114 271	8	134 348	42	34 017	372	102 088	561	23 577	18	66 322	2 170	19 463	86
proguard	130 022	256	116 381	8	136 333	40	35 155	397	102 904	584	14 966	17	20 071	20	13 050	44
sablecc	127 274	242	113 630	8	133 552	39	18 970	129	100 287	557	12 636	15	17 714	18	10 789	45
average	245.2 s		7.8 s		39.2 s		224.2 s		560.9 s		16.2 s		448.2 s		51.9 s	

4.4 Impact on Type Consumers

To answer **RQ4**, we performed a case study with OPAL's field immutability analysis [20]. It analyzes whether a field can be modified or is guaranteed to be immutable. In particular, this depends on whether the types of objects stored in the field are immutable.

We first studied OPAL's field immutability analysis to find out that it relied on the declared type of a field (i.e., it has CHA precision) plus additional ad-hoc precision improvements (we refer to this as *ad-hoc CHA*). Next, we implemented a new version of that analysis using Unimocg's type iterator interface. Our hypothesis is that the field immutability analysis benefits directly from using the type iterator in terms of both improved precision and reduced code size. Precision of the immutability analysis depends on precise type information, because `final` fields are either *transitively immutable*, if they can only refer to immutable objects, or otherwise *non-transitively immutable*, if that is not the case; *dependent immutability* (for fields with generic types) is located between these two levels. More precise type information thus allows to assign the more precise value *transitively immutable* to more fields. Code size is expected to be reduced because the immutability analysis does not need to implement (ad-hoc) logic for inferring type information.

We compare the ad-hoc CHA implementation by Roth et al. [20] with the Unimocg-based implementation using different call-graph algorithms. Previously, such an exploration of different call-graph algorithms would not have been possible as the precision was hard-coded into the analysis (ad-hoc CHA). We analyze the OpenJDK 1.8.0_342-b07 used in the previous section as OpenJDK 1.8 was also the primary evaluation target of the original implementation [20].

Table 4 shows the results concerning precision, clearly showing that using more precise call-graph algorithms (in particular, more precise type information) significantly improves the precision of the immutability analysis implemented as a Unimocg type consumer. For instance, using the RTA type iterator results in 17 604 more fields, i.e., 18.8% of all fields, found to be transitively immutable compared to CHA. Using XTA further improved the precision, with 4481 more transitively immutable fields compared to RTA. The CHA type iterator results in less precision than the baseline, because of the removed ad-hoc logic of the baseline for improving precision upon CHA. Yet, (a) the precision reduction is small (0.9% of all fields get a less precise result), and (b) one could avoid even this small imprecision (at the cost of some performance) by using Unimocg's *foreachAlloc* method instead without adding complexity to the implementation of the immutability analysis.

[7] Not analyzing the full JDK

Table 4: Field Immutability Results for OpenJDK

Algorithm	mutable	⊒ non-trans.	⊒ depen.	⊒ trans.
Ad-hoc CHA	23 195	24 296	108	46 368
CHA	23 195	25 252	20	45 500
RTA	23 195	7 352	316	63 104
XTA	23 195	2 871	316	67 585

depen. = dependently immutable, trans. = transitively immutable
Higher numbers in columns to the right = more precise

Moreover, by implementing the field immutability analysis as a Unimocg type consumer, we could replace the baseline's *ad-hoc CHA* logic (95 lines of code, or 20% of the total size of the field immutability analysis) by 26 LOCs for using the type iterator, while achieving higher precision and enabling experimentation with different call-graph algorithms. The positive effect on code quality is more pronounced than the mere numbers may suggest: The removed code was complex and a clear violation of the principle of separation of concerns, as it was not concerned with the actual task of analyzing immutability. Using Unimocg, duplication of functionality is reduced and separation of concerns is re-established.

- *Implementing the field immutability analysis as a type consumer improved it compared to an ad-hoc implementation.*
- *Its precision is always consistent with the chosen call-graph algorithm and directly benefits from more precise call-graph algorithms.*
- *It is less complex and has a better separation of concerns.*

4.5 Threats to Validity

Our choice of Soot and WALA to compare against may be perceived as a threat to the validity of our results. One may wonder about other frameworks like Doop [3] or the many compilers that internally compute call graphs. Soot and WALA represent, however, the state of the art in static analysis frameworks and offer a variety of different call-graph algorithms. Unlike them, Doop focuses on a single family of call graphs based on CFA whereas Unimocg provides consistent soundness across call-graph algorithms from different families. Thus, it is easy for Doop to provide a single representation of type information and reuse call-resolution code, which is not the

case if dissimilar families of call-graph algorithms are to be supported. Compilers usually use a single call-graph implementation that is tailored to be strictly sound but does not have to consider details of complex language features, such as reflection.

Our use of the benchmark by Reif et al. could also introduce a bias in the considered features. However, it is the only suitable benchmark available to compare call-graph soundness and it has been carefully crafted to extensively cover the relevant features for call-graph construction on the Java Virtual Machine.

5 RELATED WORK

In this section, we present related work regarding call-graph construction. We discuss general-purpose analysis frameworks and families of call-graph algorithms. Finally, we present related work about supporting complex language features in call-graph algorithms and measuring soundness w.r.t. such features.

5.1 Analysis Frameworks

OPAL [10] previously supported RTA with a high level of soundness [16]. While language features were supported by individual modules, they were not built to interpret different kinds of type information. Thus, they had a fixed level of precision and could not be reused for consistent soundness with other call-graph algorithms such as CFA. They also used ad-hoc methods of computing local allocation information to improve soundness and precision. OPAL's immutability analysis [20] also relied on such ad-hoc methods that are not needed with Unimocg.

Soot [30] supports different call-graph algorithms such as CHA, RTA, and VTA (Variable Type Analysis [28]). While CHA is implemented directly, other call-graph algorithms such as RTA and VTA are emulated in the points-to framework SPARK [13]. This allows them to reuse call-resolution code across different algorithms. However, the emulation of less precise algorithms such as RTA comes at the cost of performance as we show in our evaluation (Table 3).

The *Watson Libraries for Analysis (WALA)* [11] also support different call-graph algorithms like CHA, RTA, and CFA. WALA decouples the creation of call graphs from the call resolution for language features with the Java interfaces called *call-graph builder* and *context interpreter*. In particular, a call-graph builder computes a call graph with RTA or CFA precision, whereas a context interpreter resolves calls of built-in language features such as reflection. Crucially, unlike Unimocg, WALA does not decouple the analysis of type information. For example, the RTA call-graph builder is closely coupled to a points-to analysis to determine the instantiated classes. Also, the RTA call-graph builder implements special handling of the `clone` method and in this is strongly coupled to an interpreter for that feature. As *context interpreters* are only invoked on explicit call instructions, features such as static initialization that happen regardless of explicit calls cannot be handled. WALA's CHA implementation does not use the call-graph-builder facilities and has its own redundant implementation of some features such as invocation of static class initializers. In contrast, Unimocg decouples these concerns, leading to more consistent soundness. WALA's architecture for call-graph construction is not extensively documented and has not been discussed in a scientific publication so far.

The *Doop* framework [3] also supports points-to based call-graph algorithms. Based on Datalog, it includes rule sets for language features such as reflection. These rule sets are modularly shared between call graphs of different context sensitivity. However, popular algorithms such as CHA and RTA are not feasible with Doop: even if they were implemented using Datalog rule sets, they would need all type information for every local variable to be stored individually like in points-to based solutions in order for other rule sets like the base call graph and reflection to use them. This would result in high memory use, offsetting the benefits of simpler call-graph algorithms. Unimocg, in contrast, provides just an interface to access the type information computed anyway. For RTA, e.g., type information for a local variable is provided directly from a single set of all classes instantiated in the program. Thus, type information is stored only once, keeping memory requirements minimal.

None of the frameworks discussed here decouples the computation of type information from call resolution for dissimilar families of call-graph algorithms. In contrast, Unimocgprovides a unified interface to type information to be used by any call resolution code for any call-graph algorithm. This unified interface can additionally be used by analyses beyond call-graph construction.

5.2 Families of Call-Graph Algorithms

Class-hierarchy analysis [4] (CHA) is the simplest type-based call-graph algorithm as its call resolution depends solely on the statically declared type of the call's receiver. Bacon and Sweeney's rapid-type analysis [1] (RTA) improves over CHA by only considering subtypes that are instantiated by the analyzed program. However, these algorithms solely describe the resolution of standard virtual calls, neglecting other aspects, such as language features like reflection, which additionally affect call-graph construction.

Tip and Palsberg [29] propose a propagation-based call-graph framework, introducing four call-graph algorithms: CTA, FTA, MTA, and XTA. They attribute a call graph's precision to the number of sets used to approximate run-time values of expressions. CTA uses distinct sets for classes, MTA uses distinct sets for classes and fields, FTA uses distinct sets for classes and methods, and XTA uses distinct sets for fields and methods. Thus, the framework allows to instantiate various context-insensitive call-graph algorithms. However, the authors only discuss and evaluate standard virtual-call resolution. It remains unclear whether sharing additional modules to support other language features is generically possible.

Grove and Chambers [9] give a visualization of the relative precision and computation cost of the previously discussed and further call-graph algorithms. Moreover, they present a framework for call-graph algorithms that is parametric in the choice of context sensitivity. They distinguish three *contour selection functions* to allow varying levels of context sensitivity. Here, a contour denotes each context-sensitive version of a procedure. These functions enabled them to extend Shivers' k-CFA [23] to the more precise k-l-CFA algorithm. Thus, the framework allows for a single implementation for a range of points-to-based call-graph algorithms. However, their framework is not applicable to commonly used highly scalable algorithms such as CHA and RTA. Furthermore, forms of context sensitivity are restricted by the signatures of the four *contour key selection functions* for procedures, instance variables, classes, and

the environment (the latter being necessary for nested closures). Finally, their framework again only considers standard virtual-call resolution, but not how to combine this with additional modules to support various language features necessary for sound call graphs.

5.3 Feature Support And Soundness

In addition to the resolution of virtual method calls, a call graph highly depends on how other aspects, such as language features or APIs, are taken into account during call-graph construction. In recent years, researchers proposed approaches to specifically support language features and APIs such as reflection [15], dynamic proxies [7], serialization [21] or even new language instructions. For example, Fourtounis et al. [7] discussed how to add support for a the *invokedynamic* Java bytecode instruction introduced in Java which provides a new call instruction with user-defined semantics [19], which is highly-relevant to call-graph construction. Unfortunately, all of them are presented and evaluated in the context of specific call-graph algorithms, lacking comprehensive discussion how to generalize these concepts to other algorithms.

While researching these individual concepts is crucial to obtain sound and precise call graphs, it does not imply that they are implemented in commonly used call-graph frameworks. Sui et al. [26] compared call graphs generated by Soot, WALA, and Doop and measured their differences in soundness. Finding unsoundness, they investigated its root causes in follow up work [27]. Comparing static call graphs and dynamically recorded context call trees, they find that advanced language features, such as reflection, serialization, or native methods, are significant reasons for unsoundness.

Reif et al. [16, 17] investigated the feature support of various call-graph algorithms from Soot, WALA, Doop, and OPAL using a hand-crafted test suite. Their test suite consists of test cases, each testing whether a particular call-graph algorithm supports a specific Java language feature or API. As a result, they found that even call graphs from the same framework support different feature sets.

These studies show the need for modular call-graph construction that supports not only implementing call graphs with different precision and scalability trade-offs, but also to implement generic feature support among different algorithms as we do with Unimocg.

6 CONCLUSION

We have shown that modular call-graph construction that decouples the computation of types of local variables from the resolution of call targets is sorely needed. This decoupling enables modular composition of different analyses that contribute to both type computation and call resolution, making it possible to model different language features and APIs in individual modules. With individual modules, feature support can be implemented and reasoned about in isolation. This is necessary to facilitate support for a multitude of such features that are relevant to call-graph construction. As a result, users of call graphs can rely on consistent feature support and analysis developers can easily add new algorithms or language features while reusing existing components.

We presented our modular architecture, Unimocg, that achieves this decoupling through a unified interface, the type iterator, that can be queried by call-resolver modules to get type information from type-producer modules. This allows all call resolvers to collaborate despite being fully independent of each other. Further analyses that need type information, such as immutability analyses, can benefit from this unified interface as well. With its modular architecture, Unimocg already supports ten different call-graph algorithms from vastly different families: CHA, RTA, the XTA family including MTA, FTA, and CTA, and four k-l-CFA-based algorithms.

Our evaluation shows that Unimocg can provide consistently high soundness across different call-graph algorithms that span a wide range of precision, an important improvement over the state of the art. The consistently high soundness does not come at the price of sacrificing precision or performance. We also showed how Unimocg's unified interface can be used to improve the precision of an immutability analysis while simplifying its implementation.

While we discussed call-graph construction for JVM-based languages here, similar issues apply to other programming languages as well and Unimocg's architecture is not specific to the JVM, but can be used for call graphs in any language.

7 DATA AVAILABILITY

Raw data presented in the tables and the scripts used to generate them are provided as an artifact.[8]

ACKNOWLEDGMENTS

This work was supported by the DFG as part of CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

REFERENCES

[1] David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Jose, CA, USA) *(OOPSLA'96)*. ACM, 324–341. https://doi.org/10.1145/236337.236371

[2] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering* (Honolulu, HI, USA) *(ICSE'11)*. IEEE, 241–250. https://doi.org/10.1145/1985793.1985827

[3] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, FL, USA) *(OOPSLA'09)*. ACM, 243–262. https://doi.org/10.1145/1640089.1640108

[4] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *European Conference on Object-Oriented Programming* (Åarhus, Denmark) *(ECOOP'95)*. Springer, 77–101. https://doi.org/10.1007/3-540-49538-X_5

[5] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. XCorpus–An executable Corpus of Java Programs. *Journal of Object Technology* 16, 4 (2017), 1:1–1:24. https://doi.org/10.5381/jot.2017.16.4.a1

[6] Michael Eichberg, Florian Kübler, Dominik Helm, Michael Reif, Guido Salvaneschi, and Mira Mezini. 2018. Lattice Based Modularization of Static Analyses. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, The Netherlands) *(SOAP'18)*. ACM, 113–118. https://doi.org/10.1145/3236454.3236509

[7] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. 2018. Static Analysis of Java Dynamic Proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, The Netherlands) *(ISSTA'18)*. ACM, 209–220. https://doi.org/10.1145/3234988

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley.

[8] https://doi.org/10.5281/zenodo.10890010

[9] David Grove and Craig Chambers. 2001. A Framework for Call Graph Construction Algorithms. *ACM Transactions on Programming Languages and Systems* 23, 6 (2001), 685–746. https://doi.org/10.1145/506315.506316

[10] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. 2020. Modular Collaborative Program Analysis in OPAL. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE'20)*. ACM, 184–196. https://doi.org/10.1145/3368089.3409765

[11] IBM. 2024. WALA - Static Analysis Framework for Java. http://wala.sourceforge.net/. [Online; accessed 11-March-2024].

[12] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE'17)*. IEEE, 507–518. https://doi.org/10.1109/ICSE.2017.53

[13] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using Spark. In *Compiler Construction* (Warsaw, Poland) *(CC'03)*. Springer, 153–169. https://doi.org/10.1007/3-540-36579-6_12

[14] Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *International Conference on Compiler Construction* (Vienna, Austria) *(CC'06)*. Springer, 47–64. https://doi.org/10.1007/11688839_5

[15] Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Asian Symposium on Programming Languages and Systems* (Tsukuba, Japan) *(APLAS'05)*. Springer, 139–160. https://doi.org/10.1007/11575467_11

[16] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA'19)*. ACM, 251–261. https://doi.org/10.1145/3293882.3330555

[17] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, The Netherlands) *(SOAP'18)*. ACM, 107–112. https://doi.org/10.1145/3236454.3236503

[18] Michael Reif, Florian Kübler, Dominik Helm, Ben Hermann, Michael Eichberg, and Mira Mezini. 2020. TACAI: An Intermediate Representation Based on Abstract Interpretation. In *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (London, UK) *(SOAP'20)*. ACM, 2–7. https://doi.org/10.1145/3394451.3397204

[19] John R. Rose. 2009. Bytecodes meet Combinators: invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages* (Orlando, FL, USA) *(VMIL'09)*. ACM, 2:1–2:11. https://doi.org/10.1145/1711506.1711508

[20] Tobias Roth, Dominik Helm, Michael Reif, and Mira Mezini. 2021. CiFi: Versatile Analysis of Class and Field Immutability. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE'21)*. IEEE, 979–990. https://doi.org/10.1109/ASE51524.2021.9678903

[21] Joanna C. S. Santos, Reese A. Jones, Chinomso Ashiogwu, and Mehdi Mirakhorli. 2021. Serialization-Aware Call Graph Construction. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Virtual Event, Canada) *(SOAP'21)*. ACM, 37–42. https://doi.org/10.1145/3460946.3464319

[22] Joanna C. S. Santos, Reese A. Jones, and Mehdi Mirakhorli. 2020. Salsa: Static Analysis of Serialization Features. In *Proceedings of the 22nd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs* (Virtual Event, USA) *(FTfJP'20)*. ACM, 18–25. https://doi.org/10.1145/3427761.3428343

[23] Olin Shivers. 1988. Control Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, GA, USA) *(PLDI'88)*. ACM, 164–174. https://doi.org/10.1145/53990.54007

[24] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages of Taming Lambda*. Ph. D. Dissertation. USA.

[25] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, TX, USA) *(POPL'11)*. ACM, 17–30. https://doi.org/10.1145/1926385.1926390

[26] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. In *Programming Languages and Systems* (Wellington, New Zealand) *(APLAS'18)*. Springer, 69–88. https://doi.org/10.1007/978-3-030-02768-1_4

[27] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE'20)*. ACM, 1049–1060. https://doi.org/10.1145/3377811.3380441

[28] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical Virtual Method Call Resolution for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, MN, USA) *(OOPSLA'00)*. ACM, 264–280. https://doi.org/10.1145/354222.353189

[29] Frank Tip and Jens Palsberg. 2000. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, MN, USA) *(OOPSLA'00)*. ACM, 281–293. https://doi.org/10.1145/353171.353190

[30] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON'99)*. IBM Press, 13.

[31] Raja Vallée-Rai and Laurie J. Hendren. 1998. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. Technical Report. Sable Research Group. McGill University.

Received 15-DEC-2023; accepted 2024-03-02