Modular Collaborative Program Analysis in OPAL

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

Florian Kübler
kuebler@cs.tu-darmstadt.de
Department of Computer Science
Technische Universität Darmstadt
Germany

Michael Reif
reif@cs.tu-darmstadt.de
Department of Computer Science
Technische Universität Darmstadt
Germany

Michael Eichberg
mail@michael-eichberg.de
Department of Computer Science
Technische Universität Darmstadt
Germany

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

ABSTRACT

Current approaches combining multiple static analyses deriving different, independent properties focus either on modularity or performance. Whereas declarative approaches facilitate modularity and automated, analysis-independent optimizations, imperative approaches foster manual, analysis-specific optimizations.

In this paper, we present a novel approach to static analyses that leverages the modularity of blackboard systems and combines declarative and imperative techniques. Our approach allows exchangeability, and pluggable extension of analyses in order to improve sound(i)ness, precision, and scalability and explicitly enables the combination of otherwise incompatible analyses. With our approach integrated in the OPAL framework, we were able to implement various dissimilar analyses, including a points-to analysis that outperforms an equivalent analysis from Doop, the state-of-the-art points-to analysis framework.

CCS CONCEPTS

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

KEYWORDS

Static Analysis, Blackboard System, Modularization, Composition, Parallelization

ACM Reference Format:
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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368089.3409765

1 INTRODUCTION

Solving complex static analysis problems often involves solving several distinct but interdependent sub-problems. Analyzing a method's purity, e.g., involves interdependent mutability sub-analyses of classes and fields [38, 40, 64] as well as an escape analysis [15, 50].

Traditionally, static analyses have been implemented in an imperative monolithic style, i.e., one super-analysis computes the results of all sub-problems. Not only do monolithic designs become complex when mutually dependent problems are involved [12]. More importantly, individual sub-analyses cannot be developed in isolation, cannot be reused for other analyses, and cannot easily be added, removed, and exchanged to trade-off between precision, sound(i)ness [55], and performance in a fine-tuned way, i.e., to enable pluggable precision/sound(i)ness/performance.

To address these requirements, it is desirable to encode solutions for sub-problems of a complex static analysis in separate modules. However, while encoded in independent modules, the execution of inter-dependent sub-analyses needs to be interleaved to enable exchanging intermediate results. The latter is often necessary for optimal precision, as has been proven by the theory of reduced products in abstract interpretation [19] and was more recently demonstrated for other kinds of analyses [11, 28, 38].

Recently, declarative approaches to static analysis using the Datalog language [12, 56, 78] are gaining increased popularity—especially in the area of points-to analyses [12, 72, 74, 78]. Such approaches nicely support the requirements stated above. Analyses are implemented as sets of rules that are evaluated by an underlying constraint solver. Thus, complex analyses can be broken down into simpler, independently-developed analyses. The underlying solver transparently resolves their dependencies and propagates intermediate updates according to the specified rules, thus enabling interleaved execution. Moreover, the solver can (a) apply analysis-independent optimizations, e.g., by rearranging the computation order (although manual optimization is still necessary [12, 71]), and/or (b) automatically parallelize the execution [45].

However, using Datalog and giving solvers full control comes with *drawbacks in terms of both performance and generality.* First, it is not possible to exploit analysis-specific knowledge in managing the execution and dependencies of the analyses. Such knowledge can help boost scalability. For example, an imperative purity analysis that determines whether a method is deterministic by, among others, checking the mutability of fields $f_1, ..., f_n$ could drop further checks as soon as any f_i is found to be mutable. A declarative analysis whose execution is driven by a general-purpose solver cannot take this short-cut. Analysis-specific knowledge is also valuable to correctly compose incompatible optimistic and pessimistic analyses (as defined in [34, 53]). Second, the Datalog solver uses analysis-independent data structures and analyses cannot exploit data structures that are tailored for their specific needs. Such optimized data structures, like tries, can be crucial for achieving performance. Finally, the fully declarative approach fosters a one-size-fits-all style, limiting generality. For instance, by relying on relations, Datalog-based approaches support only set-based lattices, while many common analyses require other kinds of lattices. Constant propagation, e.g., is usually implemented via singleton-value-based lattices, making it infeasible to implement it using Datalog [56, 73].

In this paper, we address these issues of declarative approaches, without comprimizing on their benefts. Specifically, we propose a novel generic approach together with a proof-of-concept implementation in the OPAL framework [26] for lattice-based fixed-point computations with support for lattices of any kind including singleton-value-based, interval, and set lattices. Like fully declarative approaches, it features modular analyses encoded as independently compilable, exchangeable, and extensible units. However, it does not rely on a general-purpose declarative framework and constraint solver. It offers a specialized approach mixing imperative and declarative styles. The developer of an OPAL analysis implements its core functionality imperatively, but declaratively specifies its dependencies, e.g., the lattice that the analysis computes and lattices it depends on to do so, as well as several constraints regarding its execution. Dependencies and constraints are automatically handled by our custom solver during analysis execution.

Our architecture is reminiscent of blackboard systems [17]: Dependent analyses implemented in decoupled modules coordinate their executions implicitly by writing into and reading from a central data structure (the "blackboard"). Whenever new (intermediate) results are written to the blackboard, the solver automatically (and concurrently) schedules the execution of dependent analyses to satisfy the declaratively specified dependencies and constraints.

Like declarative approaches, we decouple mutually dependent analyses, enabling their isolated development and interleaved parallel execution out-of-the-box. At the same time, we improve over declarative approaches in two regards. First, beyond automatic and transparent optimizations and parallelization, by featuring an imperative programming style within each analysis module, our approach enables analysis-specific optimizations and data structures. The possibility to specify fine-grained (analysis-specific) constraints enables further optimizations, e.g., suppressing interleaved execution of some analyses to avoid unnecessary intermediate computations. Second, with a custom solver that is agnostic of the lattices used by analyses, our approach is generic and supports arbitrary kinds of analyses. Using it, one can naturally express dataflow and constraint-based analyses based on arbitrary lattices. Moreover, declarative declarations enable OPAL to consider analysis-specific constraints in managing dependencies. To the best of our knowledge, this is the first approach to correctly compose lazily computed incompatible optimistic and pessimistic analyses.

To recap, this paper contributes:

- A list requirements on frameworks for collaborative static analysis that is distilled from three case studies (Section 3).
- A novel approach, that satisfies all these requirements (Section 4). It advances the state-of-the-art in implementing modular inter-dependent analyses.
- A thorough evaluation of the approach that supports our claims on generality, showcases its modularity features, points out performance improvements over Doop [12], the state-of-the-art declarative framework, and provides promising results for parallelization (Section 5).

2 BACKGROUND AND TERMINOLOGY

In this section, we shortly introduce blackboard systems and present terminology used throughout the paper.

Blackboard Systems [17] use a central data structure - *the blackboard* - to coordinate the collaborative work of otherwise decoupled *knowledge sources*. The latter contribute (partial) information to the blackboard towards collaboratively reaching an overall goal. The blackboard notifies knowledge sources about availability of new information they might require through a control mechanism that decides which knowledge sources should be executed in what order. The information can then be queried by the knowledge sources, which execute and produce further information. Each execution of a knowledge source is called an *activation*.

Entity: The term is used to characterize anything one can compute some information for. Entities can be concrete code elements, e.g., classes, methods, or allocation sites, or abstract concepts such as all subtypes of a class. The set of entities is not necessarily defined a priori and can be created on-the-fly while analyses execute.

Property Kind: The term characterizes a specific kind of information that can be computed for an entity, e.g., mutability of classes, purity of methods, or callees of a specific method. Each property kind represents one lattice of possible values.

Property: The term characterizes a specific value in the lattice of some property kind that is attached to some entity, e.g., a class can be mutable or immutable, a method can be pure or impure, a specific method may invoke a specific set of methods. Per entity at most one property of a specific kind can be computed.

Analysis: The term characterizes an algorithm that given an entity computes its property of a certain kind. We say that *an analysis computes a property kind* as shorthand for "an analysis computes properties of that property kind for a given kind of entity". Analyses are knowledge sources in the sense of the blackboard architecture; the properties they compute constitute the information that they contribute to and/or query from the blackboard.

3 CASE STUDIES

We discuss case studies involving several interrelated sub-analyses to distill a list of requirements on static analysis frameworks. During the discussion, we *emphasize* concepts whenever they occur. The case studies represent very dissimilar kinds of analyses. In particular, they require different kinds of lattices, including singleton-value lattices (e.g. in 3.3) and set-based lattices (e.g. in 3.2). This motivates the first requirement: Static analyses frameworks must support varied domain lattices (**R1**).

3.1 Three-Address Code

The first case study is an analysis to produce a three-address code representation (TAC) of JVM bytecode, presented in more detail in previous work [66]. In its basic version, TAC uses def/use, type, and value information (including constant propagation) provided by an abstract-interpretation-based analysis (AI). To increase precision, AI may be enhanced with analyses that refine type and the value information for method return values and fields. However, such additional analyses may negatively affect the runtime. Hence, systematic investigation of the precision/performance trade-off is needed to decide whether to use such additional analyses on a case-by-case basis. To this end, a separation into modules that can be enabled/disabled is beneficial. In general, we derive the following requirements regarding support for modular pluggable analyses.

For systematically studying precision/soundness/performance trade-offs, static analysis frameworks should support en/disabling inter-dependent analyses (**R2**). To maximize pluggability, analyses should be defined in decoupled modules, and yet be able to collaboratively compute properties (*collaborative analyses*). As individual analyses can be disabled, it should be possible to specify soundly over-approximated *fallback values*[1] for the properties they compute, to be used by dependent analyses in lack of actual results (**R3**).

Moreover, an approach for modular collaborative analyses should support their *interleaved execution* without them knowing about each other's existence (**R4**). Two analyses are executed interleaved, if they can interchange *intermediate results*. This is important for optimal precision [19]: knowledge gained during the execution of analysis A_1 may be used by the execution of another analysis A_2 on-the-fly to refine its result and, in turn, this may enable further refinement for A_1. The precision of field- and return-value refinement analyses profits from interleaved executions, as they depend on each other cyclically. If a method `m` returns the value of a field `f`, then `m`'s return value depends on `f`'s value. If the value returned by `m` is written into `f`, then `f`'s value also depends on `m`'s return value.

However, interleaved execution must in specific cases be suppressed to ensure correctness. This is the case for the composition of *pessimistic* and *optimistic* analyses. Pessimistic analyses start with a sound but potentially imprecise assumption and eventually refine it. Optimistic analyses start with an unsound but (over)precise assumption and progress by reducing (over)precision towards a sound result. Field- and return-value refinement analyses are pessimistic—the declared return type of method *m*, say `List`, is a sound but eventually imprecise initial value for the return-value analysis; during the execution, the analysis may find out that *m* actually returns the more precise result, say `ArrayList`. AI is an optimistic analysis—it starts with the unsound assumption that all code is dead and refines it by adding statements found to be alive towards a sound, but potentially less precise result. Optimistic and pessimistic analyses are *incompatible* for interleaved execution, because they refine the respective lattices in opposite directions. As a result, exchanging intermediate results may cause inconsistencies, thereby violating monotonicity. Thus, the analysis framework must enforce that only *final results* of pessimistic analyses are passed to dependent optimistic analyses (and vice-versa), avoiding interleaving and *suppressing* non-final updates (**R5**).

For illustration, consider the example of some piece of code, say *c*, that contains a call to a method m_1 that is mutually recursive with a method m_2, but is conditioned on a field f being an instance of `LinkedList`. To analyze *c*, we combine a field-value analysis *FA*, an *AI* analysis, and a call graph construction algorithm, *CG*. Assume that *FA*, which is a pessimistic analysis, initially reports the type of the field f to be `List`. Given this information, AI would optimistically consider *c* to be live and *CG*, hence, will consider both m_1 and m_2 to be reachable. Because of the mutual recursion (and also because of the monotonicity requirement), this result cannot be changed later, if *FA* finds out that f can only contain `ArrayLists`. If, however, the latter information was present when AI analyzed the code, *c* would have been marked as dead, and *CG* would have marked m_1 and m_2 as unreachable. Thus, the results of this combination of analyses is non-deterministic and possibly incorrect (imprecise, if m_1 and m_2 are falsely reported to be reachable).

3.2 Modular Call Graph Construction

Inter-procedural analyses presume a call graph (CG): Given method `m`, CG provides information about (a) methods that may be invoked at a call site in `m` (callees) and (b) call sites from which `m` may be invoked (callers). We use the CG to motivate the need for supporting further kinds of execution interleaving (beyond **R4**) as well as further requirements. The previous case study illustrated the need for interleaved execution of inter-dependent analyses that calculate different properties and operate on different entities (composition of analyses for refining field and return values with TAC). The CG use case illustrates two further kinds of interleaved execution.

First, we need interleaved execution of multiple instances of the same analysis operating on different code entities to collaboratively compute a single property, whereby each instance contributes partial results (**R6**). For example, different executions of a CG analysis for different callers of a method *m* need to contribute their *partial results* to collaboratively derive all of *m*'s callers (computing callers of a method is inherently non-local).

Second, we also need to support interleaving of independent analyses that collaboratively compute a single property (**R7**). Consider, e.g., the computation of the callees of `m`. A CG analysis can in principle consider *m* in isolation. A monolithic analysis for callees is nonetheless not suitable. It makes sense to distinguish between one sub-analysis that handles standard invocation instructions (e.g., CHA [23], RTA [4], points-to-based [12] analysis) and sub-analyses dedicated to non-standard ways of method invocation through specific language features, e.g., reflection, native methods, or functionality related to threads, serialization, etc. Non-standard invocation requires specific handling (e.g., one may deliberately not want to perform reflection resolution, or may want to perform it based on dynamic execution traces). By offering such specialized analyses as decoupled modules, they become highly reusable and can be combined with different call-graph analysis for standard invocation

[1]To minimize the effect of fallback values on precision, it makes sense to compute the fallback by using locally available information, e.g., using declared type information, instead of always returning the same over-approximated value.

instructions. This makes the call graph construction highly configurable for fine-tuning its performance and sound(i)ness. Hence, not only a method's callers but also its callees need to be computed collaboratively. This time, different analyses targeting different language features, rather than different executions of the same CG analysis, contribute to the same property.

Handling special language features may even rely on integrating results of external tools or precomputed values (**R8**). For instance, one may choose to integrate the results of TamiFlex [10] for reflective calls, or external tools for analyzing native methods.

The CG case study also motivates support for specifying precise *default values* (**R9**) (in addition to sound fallback values). Consider the case of an unreachable method *m*. The CG analysis will never compute callees or caller information for *m*. However, this lack of results is an inherent property of the entity and not the result of a missing/disabled analysis. A sound fallback value to compensate the deactivation of the CG module for *m* may have to include all methods and hence be too imprecise. Instead, analyses depending on the CG should get the information that *m* is unreachable—the precise default value. The analysis developer knows such information and should be enabled to tell the framework.

Another requirement is motivated by the CG. The CG construction unfolds along the transitive closure of methods reachable from some entry points. Hence, it does not make sense to execute the decoupled modules collaboratively constructing the CG—each handling a particular language feature—globally on all methods of a program. Instead, they should be *triggered* only when the overall analysis progress discovers a newly reachable method. Hence, the framework must support triggering analyses once the first (intermediate) result for a property is recorded (**R10**).

Our previous work [65] provides empirical evidence that encoding an RTA sub-analysis and sub-analyses for language-specific features as collaborative interleaved modules, results in more sound call graphs and better performance compared to call graph analyses of the Soot [76], WALA [41], and Doop [12] frameworks.

3.3 Mutability, Escape, and Purity Analysis

The analyses in this subsection illustrate the need for further kinds of activation modes in addition to triggered analyses, illustrated in the previous subsection: (a) *eager analyses*, which refers to computing an analysis for all entities in the analyzed program, and (b) *lazy analyses*, i.e., executing an analysis A_1 only for the entities for which the property that A_1 computes is queried by some (potentially the same) analysis A_2. A further requirement shown by analyses in this subsection is that the framework should allow analyses to enforce an execution order that overrides the one determined by the solver.

The use case involves analyses for method purity, class and field mutability [40, 64], and escape information [15, 50]. The latter includes aggregated information on field locality and return-value freshness (cf. [38]). The analyses in this case study interact tightly and compute properties that may be relevant for both end users (e.g., method purity) and further analyses (e.g., escape information). Complex dependencies exists between all these analyses. To fine-tune the precision/performance trade-off, several analyses for these property kinds with different precision can be exchanged as needed; all are *optimistic* and use TAC and/or the CG information.

Table 1: Summary of Requirements

Lattices and values	
R1	Support for different kinds of lattices (3.1, 3.2, 3.3)
R3	Fallbacks of properties when no analysis is scheduled (3.1, 3.3)
R9	Default values for entities not reached by an analysis (3.2)
Composability	
R2	Support for enabling/disabling individual analyses (3.1, 3.2, 3.3)
R4	Interleaved execution with circular dependencies (3.1, 3.2, 3.3)
R5	Combination of optimistic and pessimistic analyses (3.1)
R6	Different activations contributing to a single property (3.2)
R7	Independent analyses contributing to a single property (3.2)
Initiation of property computations	
R8	Precomputed property values (3.2, 3.3)
R10	Start computation once an analysis reaches an entity (3.2)
R11	Start computation eagerly for a predefined set of entities (3.3)
R12	Start computation lazily for entities requested (3.1, 3.3)
R13	Start computation as guided by an analysis (3.3)

Since the results of analyses in this case study may be of interest to the end user, it is useful to compute them for all possible entities eagerly (**R11**), e.g., computing the mutability of all fields in the program. However, when the field mutability is only used to support, e.g., the purity analysis, it may be beneficial for performance reasons to compute it lazily (**R12**), i.e., only for the fields for which mutability is queried by the purity analysis. This illustrates that we need both eager and lazy execution modes. Eager and lazy versions of the same analysis should typically share the code and only be registered with the framework in different ways. The class mutability analysis also illustrates the need to configure the framework with analysis-specific execution orders (**R13**): For performance reasons, it makes sense to analyze classes in a program in a top-down order starting with parent classes before their children.

Our previous work ([38]) provides empirical evidence for the requirements stated in this section. An implementation of the purity sub-analysis of this case study (and through transitive use, the mutability and escape sub-analyses) as collaborative analyses with interleaved execution showed higher precision, more fine-granular results and similar performance characteristics compared to the then state-of-the-art purity inference tool ReIm [40].

3.4 Interim Summary

Table 1 summarizes the requirements along the case studies motivating them. Existing frameworks do not satisfy all of them. Imperative frameworks lack support for modularity, especially **R5**, **R6**, and **R7**. Declarative approaches, e.g., Doop [12], have other limitations: Being bound to relations for modeling properties, they can not express the range of different analyses represented by our case studies (**R1**). They also fail to support sound interactions between incompatible analyses (**R5**). By giving the solver full control, they do not support different analysis-specific activation modes (**R10**-**R13**).

4 APPROACH

OPAL is the first static analysis framework to build upon the concept of blackboard systems: Static analysis modules correspond to

```
sealed trait ClassMutability extends PropertyKind {
  def fallback(Type theClass) = MutableClass
}
case object ImmutableClass extends ClassMutability
case object MutableClass extends ClassMutability
```

Listing 1: Class Mutability Lattice

knowledge sources; the store that manages the computed properties corresponds to the blackboard. OPAL combines imperative and declarative programming styles for analyses. The developer of an analysis A: (a) implements the lattice representation of the property values computed by A (4.1), (b) implements two *imperative* functions - so-called *initial analysis function* (IAF) respectively *continuation function* (CF) (4.2), (c) declares the property kinds computed by A and properties A depends on (4.3), and (d) defines how A's results are reported to the blackboard (4.4). Guided by the declared dependencies, the blackboard and its fixed-point solver coordinate the execution of the analyses, thereby (e) ensuring all execution constraints (4.5), (f) performing fixed-point computations, whenever circular dependencies are involved (4.6), and (g) automatically scheduling and parallelizing the execution of analyses (4.7).

4.1 Representing Properties

Values of a property kind constitute a lattice structure. OPAL supports singleton value-based, interval, or set-based lattices are possible (**R1**). A lattice's bottom value models the best possible value (e.g., pure for method purity); its top value the sound over-approximation (e.g., impure). Lattices must satisfy the ascending (descending) chain condition to ensure termination of optimistic (pessimistic) analyses. When defining a property kind, developers can choose the most suitable data structures for efficiency.

Developers can also specify *fallback* and *default* values. The blackboard will return the *fallback value* for some requested property, p of kind k, if no analysis is available for k (**R3**). As it is a sound over-approximation, the lattice's top value is a good choice - however, the fallback value can also be provided by a "proxy" analysis function that does not query the blackboard, avoiding cyclic dependencies. The blackboard will return a *default value* for p, if an analysis is available, but did not produce any result for some entity (**R9**). For instance, call graph analyses only examine methods reachable from entry points - for any non-reachable method, m, a default value can be used to state that m is dead and has no (relevant) callees. A sound fallback value would include all possible methods as callees of m; thus, in this case, the default value provides more information than a fallback value. If no default value is declared, the fallback value is returned.

Developers implement property kinds by specifying an interface, which can be used to access and manipulate the property values. When the PropertyKind trait is extended, the framework assigns an identifier, which can be used to query the blackboard for properties of that kind. Listing 1 shows exemplary Scala code of a simple class mutability property kind. Lines 1 to 3 define the base trait for the property kind and give a sound fallback value in line 2. The two possible property values are defined in lines 4 and 5.

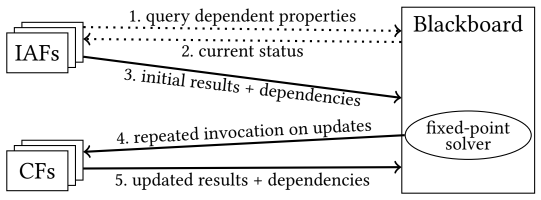
IAFs
CFs
Blackboard
fixed-point solver
1. query dependent properties
2. current status
3. initial results + dependencies
4. repeated invocation on updates
5. updated results + dependencies

Figure 1: Overview

4.2 Analysis Structure

An overview of OPAL's analysis structure is shown in Figure 1. As mentioned, the analyses are structured in two parts: An *initial analysis function (IAF)* and one or more *continuation functions (CFs)*. These functions can be implemented in any way, as long as they provide their results as defined by the property kind.

For each entity e to be analyzed by A, A's IAF is executed. The IAF collects information directly from e's bytecode in order to compute its result. If it needs additional information pertaining to some other entity e or from another analysis that computes a property kind k, the IAF queries the blackboard for these dependencies, using the identifiers of e and k to find the relevant information (arrow 1. in Figure 1). The blackboard will return the currently available value (2.). This value may, however, not be available, or not final, either because the respective analysis was not yet executed or because it has dependencies that yet need to be satisfied. Once the IAF completes analyzing the entity, it returns to the blackboard (a) a result computed based on the currently available information and (b) any remaining dependencies, along with a continuation function (CF) (3.). Similar to the solver of *declarative* frameworks, the blackboard resolves dependencies and automatically invokes the CFs whenever updates to these dependencies become available (4.). On completion, CFs also return their updated results to blackboard (5.), potentially triggering the execution of other CFs. While the IAF is written imperatively (dotted queries in Figure 1), the subsequent execution is performed similar to declarative frameworks (straight lines) by having results declare their dependencies and the solver being responsible to satisfy them. Executions of the IAFs and CFs are called *analysis activations*. To ensure determinism, OPAL executes the activations for a single property sequentially, while IAFs and CFs for other properties can execute concurrently.

As analyses get notified about dependency updates through the invocation of the CF, it is not necessary that dependencies are computed before or when they are queried. Instead, they can be computed asynchronously and lazily, i.e., on-demand (**R12**). This also allows OPAL to handle cyclic dependencies (**R4**).

Apart from adhering to this basic structure, developers may use any suitable strategy to implement an analysis A. A may, e.g., focus on specific statements instead of traversing the entire code of a method (OPAL provides pre-analyses to query specific parts of the code, e.g., all statements that access a specific field). Also, analyses can internally use any data structure suitable to achieve good performance. For illustration, Listing 2 shows an excerpt from a simple class mutability analysis' initial analysis function. The IAF is given the entity to analyze (Line 1). Lines 3 to 7 show how to retrieve and handle properties required to compute the IAF's result:

```
1  def analyze(Type theClass) = {
2    [...]
3    Blackboard.get(field, FieldMutability) match {
4    case _: MutableField => return Result(theClass, MutableClass)
5    case dependee: ImmutableField =>
6      if (!dependee.isFinal) dependees += (field -> dependee)
7    }
8    [...]
9    Result(theClass, ImmutableClass, dependees, continuation)
10 }
```

Listing 2: Class Mutability Analysis

The required property (the mutability of an instance field of the analyzed class) is queried from the blackboard (line 3) and based on the returned value, the IAF may compute its result (as in line 4) or keep the dependency in a list of dependees (line 6) to return it alongside an intermediate result later (line 9). Line 9 also specifies the continuation function to be invoked when any of the properties in `dependees` is updated. We do not show the code for that CF here, as its implementation is very similar to lines 4 to 9, i.e., based on the updated value, the (intermediate) result of the CF is determined.

There are two semantic constraints that the implementations of the analyses must satisfy, though. First, they must ensure *monotonicity of result updates* according to the used lattice. Analyses that optimistically start at a lattice's bottom value may only refine approximations upwards; pessimistic analyses only downwards. OPAL can automatically check the monotonicity of updates. Monotonicity allows analyses to know which refinements of intermediate results are still possible. Second, analyses must be *scheduling independent*: Whenever they receive the value of some other property they depend on, they must use the information provided by that value to compute the result of the current activation, i.e., they may not defer the incorporation of the newly gained information to a later activation of a continuation function. This ensures that all available information is used independent of whether the continuation is later scheduled for execution - an activation may never occur in case of cyclic dependencies. For example, once the mutability analysis of a class `C` knows that `C`'s instance field `f` is mutable, it may no longer report that `C` could be immutable. The developer of some analysis `A` must ensure that `A` is scheduling independent.

4.3 Declarative Specifications

On top of the IAF and CF, the developer of an analysis `A` specifies (a) the property kinds computed by `A`, (b) its dependencies, (c) on which entities `A` will be executed and (d) when the blackboard should start `A's` execution. These specifications are evaluated when the analysis is registered with the blackboard, before the latter takes over control of analysis activation. When registering analyses, developers may also report precomputed values to the blackboard (**R8**).

The specification of the computed property kinds also states whether intermediate results are optimistic or pessimistic and whether the analyses contributes to a collaborative computation or intends to be the only analysis computing the specified property kinds. Dependency specifications state other property kinds

```
1  override def derivesLazily = Optimistic(ClassMutability)
2  override def uses = Set(Optimistic(FieldMutability))
3  override def register() = {
4    Blackboard.set(Type.Object, ImmutableClass)
5    val analysis = new ClassMutabilityAnalysis
6    Blackboard.registerLazyAnalysis(this, analysis.analyze)
7  }
```

Listing 3: Registration of Class Mutability Analysis

on which `A` depends (which `A` queries) and whether `A` can process optimistic/pessimistic intermediate values or final values only.

Analyses can eagerly select a set of entities (e.g., all methods of the analyzed program) if it is likely necessary to perform the analysis for all of these entities (**R11**). This is, e.g., useful for analyses that are of interest to the end user, e.g., if the user is interested in the purity of all methods. Alternatively, analyses can be registered to be invoked lazily [9, 42]. Lazy analyses only compute a property if that property is queried (**R12**) by another analysis or by the end user. Finally, an analysis can specify a property kind k such that it is started for every entity for which k has been computed (**R10**).

Some analyses benefit from enforcing a specific order for computing the properties for different entities (**R13**). For instance, the class mutability analysis benefits from traversing the class hierarchy downwards, such that results for a parent class are available before any subclass is analyzed. In OPAL, this is supported by enabling the developer of an analysis `A` to declare a number of computations to be scheduled whenever `A` returns a result to the blackboard.

For illustration, Listing 3 shows the registration code for a class mutability analysis. Line 1 declares that the analysis optimistically and lazily derives class mutability. Line 2 declares that in performing its computation, it may require field mutability and that it can handle intermediate results for this property if they were computed optimistically. This declaration is complete: No property kinds other than field mutability (and class mutability) may be queried by this analysis. Line 4 registers a predefined value stating that the base class `Object` is immutable (**R8**). The IAF `analyze` is registered as a lazy analysis in line 6, i.e., the mutability of a certain class will only be computed on demand, e.g., when a purity analysis queries it.

4.4 Reporting Results

As already mentioned, analyses write intermediate and final results to the blackboard. They can report results for each single entity individually or for multiple entities at the same time. A result consists of a single lattice value representing the new value for the property or of an update function (UF) for updating the property's current value (as recorded in the blackboard) to incorporate the new result.

A UF is used for properties whose computation is not localized to a specific part of the program, e.g., the callers of a method. For such properties, constraint-based analyses [1, 59] have been used in the past; declarative analyses also provide such updates, called deltas, that only specify the change to the property value instead of the full new property value. The `UF` merges the results of one activation to the current state of the property (e.g., add a new caller to an existing set of callers). This way, activations of one or of different analyses can collaboratively contribute to a property (**R6**, **R7**).

4.5 Execution Constraints

Once the end user chooses a set of analyses to be executed (**R2**), OPAL uses the declarative specifications (Section 4.3) to check and automatically enforce restrictions on analyses that can be executed together. First, it ensures that any property kind is computed by at most one analysis or collaboratively; this is to avoid that conflicting results are reported to the blackboard. Second, if several analyses derive a property kind collaboratively, OPAL ensures that they are all either optimistic or pessimistic. Finally, OPAL ensures that all property kinds required by any analysis are derived by another analysis or there is a fallback value provided; this is to ensure that dependencies can be satisfied.

OPAL's blackboard may run optimistic and pessimistic analyses simultaneously. But, when doing so, it ensures that no intermediate results are propagated between them (**R5**). Given property kind p that is computed optimistically and pessimistic analysis A depending on p, OPAL does not forward any intermediate values of p to A's CF. The latter is triggered only when a value of p is submitted marked as final. We say that the dependency of A on p is *suppressed*. There are subtle interactions between dependency suppression and cyclic and collaborative computations, which we explain next.

First, there can be no cyclic dependencies between pessimistic and optimistic analyses. The correctness of cyclic dependency resolution relies on the assumption that all intermediate approximations have been processed and no further updates to any property involved in the cycle may happen (cf. Section 4.6). This obviously is not the case when updates are suppressed.

The interaction between dependency suppression and collaboratively computed properties is more involved. Assume a collaboratively computed property p_1 that (transitively) depends on another collaboratively computed property p_2 and consider the case when one or more of the transitive dependencies between them is suppressed[2]. In this case, OPAL must ensure that p_2's values are committed as final before p_1's values can be committed as final, too. This ensures that final values have been propagated along the suppressed dependencies. To this end, OPAL derives a *commit order* when checking the execution constraints before executing analyses. The commit order is a partial order between collaboratively computed property kinds: p_1 must be finalized later than any other collaboratively computed property kind p_2 on which p_1 depends when there is suppression between them.

Suppression of intermediate updates can also be used to improve performance: Consider the relation between TAC and AI. Both are optimistic and TAC could use intermediate AI results. But these results are typically not useful, hence, it can be beneficial to use suppression to avoid costly computation of these intermediate results and instead compute the TAC only once on the final AI result.

4.6 Fixed-Point Computation

Computation is started for the entities selected by eager analyses (**R11**) (cf. Section 4.3). Whenever intermediate values for properties are submitted, the blackboard schedules activations of continuation functions, distributing updated results to analyses that depend on them. Additionally, the blackboard starts new computations by invoking the initial analysis function for properties that are requested lazily (**R12**), are triggered by some analyses reaching a certain entity (**R10**), or whenever it is guided to do so by running analyses (**R13**). This process of scheduling IAF and CF activations is performed until no further updates are generated – the blackboard has reached a *quiescent* state. At this point, however, the properties' values may not necessarily be final, as there still may be unresolved dependencies. There are three cases to be considered.

First, an analysis was scheduled for some property kind p, but it did not analyze some entity e, for which p was requested, e.g., because e was not reachable in the call graph. In this case, the *default value* (**R9**) is inserted, which may trigger further computations, until quiescence is reached again.

Second, properties that cyclically depend on each other are not finalized yet. If such properties form a *closed strongly connected component*, i.e., they do not have any dependees outside of the cycle (but other properties may still depend on them), they are now finalized to their current value. By requiring analyses to report their results in a monotonous and scheduling independent way (cf. Section 4.2), OPAL guarantees that the cycle resolution is deterministic and sound. Again, further computations may arise from resolving cyclic dependencies (including supplying more default values and resolving further cycles) until quiescence is reached again.

Finally, the blackboard finalizes values for collaboratively computed properties. It respects the *commit order* computed previously (cf. Section 4.5): After finalizing a set of collaboratively computed properties, computation is resumed again. Only once quiescence is reached again, the next property kinds, as given by the commit order, are finalized. This is repeated until all collaboratively computed properties have been finalized.

4.7 Scheduling and Parallelization

Blackboard systems require a control component that, upon updates of the blackboard, decides which knowledge sources to activate next. In our case, this control component determines the order in which activations of dependent analyses are executed and is called *scheduler*. The order in which dependent analyses are activated can have significant effects on performance [69].

OPAL allows for the scheduler to be easily exchanged in order to select the best performing one for any chosen set of analyses. Apart from general strategies such as first-in-first-out, more specific algorithms may use the dependency structure or the values of intermediate approximations to decide the scheduling order. This is similar to the control component of blackboard systems asking knowledge sources for an estimated information gain (cf. [17]).

Blackboard systems lend themselves well to parallelization. The individual knowledge sources, i.e., analyses in our case, are decoupled and their activations (both the initial analysis and the continuations) can be executed in parallel on multiple threads. Updates to the blackboard, on the other hand, can be synchronized on a special thread or, if that becomes a bottleneck, distributed to several threads based on the property kind and/or entity. A simple implementation may consist of several threads that use a shared data structure holding the property data and use locks or other mechanisms to synchronize accesses to this shared storage.

[2]On a chain of dependencies, more than one may be suppressed. Also, if p_1 depends on p_3 and p_4 and each of those depends on p_2, there is more than one path between p_1 and p_2, on which dependencies may get suppressed.

4.8 Summary

Our approach fosters strong decoupling of reified lattices (choice of data structures), analyses (choice of algorithm), and the solver infrastructure (the concrete fixed-point solving implementation). This enables exchanging and optimizing these parts independently. As reified lattices are the basis for all communication between analyses, different versions of analyses can be implemented at different trade-offs. The solver manages execution of analyses, tracks dependencies and propagates updates, performs monotonicity checks, and computes the fixed-point solution.

5 EVALUATION

We evaluate our approach by answering the following questions:

RQ1 Does OPAL support modularization of a broad range of static analysis kinds with varying requirements?

RQ2 Does exchangeability of analysis modules benefit the end user and the developer?

RQ3 Can the framework be parallelized?

RQ4 What is the benefit of analysis-specific data structures?

RQ5 How does the performance of OPAL's analyses compare to state-of-the-art declarative approaches?

We implemented our approach on top of the Scala-based OPAL framework for JVM bytecode analysis [26]. However, the approach is framework and language independent. We answer the above research questions using the case studies of Section 3 to analyze the DaCapo 2006 benchmark [7]. We choose DaCapo because Doop, which we compare to in Section 5.5, has special support for it. Both the implementation of OPAL as well as the case studies are available in the OPAL GitHub repository[3].

All measurements were performed in a Docker container[4] on a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. Analyses were run using OpenJDK 11.0.5+10 (64-bit) with 32 GB of heap memory and Scala 2.12.9. Experiments were run seven times and we report their median runtime. We report only excerpts of the results here[5].

5.1 Support for Various Analyses

To answer **RQ1**, we implemented the case studies from Section 3 using OPAL and argue that these are representatives of different analysis kinds. The first case study represents pessimistic analyses in the context of improving precision of a three-address code representation (TAC)—it shows how basic analyses can be extended by analyses that are specialized to increase the precision of sub-problems' solutions. The modular call graph of the second case study involves tightly interacting yet decoupled analyses (e.g., points-to and call graph) and demonstrates how one can plug in further modular analyses that handle special cases of Java in order to increase the call graph's soundness. The third case study introduced several exchangeable analyses for different high-level properties (immutability, escape information, purity). The individual analyses are relatively simple and can focus on their respective property, but by using the results of other analyses, they can be more precise than a corresponding monolithic analysis of medium complexity.

[3]https://github.com/stg-tud/opal
[4]https://doi.org/10.5281/zenodo.3872848
[5]The entire results can be found here: https://doi.org/10.5281/zenodo.3972736

Table 2: Purity results for different configurations (hsqldb)

Configuration	#Pure	#SEF	#Other	#Impure	⏱ Analysis
$PA_2/FMA_1/E_1$	417	482	245	2 635	2.42 s
PA_2/E_1	363	536	245	2 635	2.40 s
$PA_2/FMA_1/E_0$	417	481	241	2 640	1.93 s
PA_2	362	504	225	2 688	0.98 s
PA_1/FMA_1	415	431	0	2 933	0.93 s
PA_0/FMA_1	104	0	0	3 675	0.70 s
PA_0	100	0	0	3 679	0.13 s

As discussed in Section 3, to achieve this modularity, several requirements need to be satisfied (cf. Table 1). Section 4 already explained how OPAL supports all of them. On the contrary, as we argue in Section 3.4, no current imperative or declarative framework supports all these requirements.

We additionally implemented a solver for *inter-procedural, finite, distributive subset problems* (IFDS) [68], a well-known general framework for dataflow problems based on graph reachability. Similar to other IFDS solvers, e.g., Heros [8], users provide a domain for their dataflow facts and four flow-functions that together specify the IFDS problem. The solver starts one computation per pair of method and entry dataflow fact and these tasks need to communicate their results. We chose IFDS as it is a general framework that allows implementing many dataflow analyses and it is dissimilar from the three case studies' analyses. In particular, it shows OPAL's support for implementing general solvers as individual analyses.

■ *OPAL's programming model enables the implementation of dissimilar analyses, fostering their modularization into a set of comprehensible, maintainable, and pluggable units. OPAL is the only static analysis framework satisfying all requirements from Section 3.4.*

5.2 Effects of Exchangeability of Analyses

Our approach strictly decouples property kinds from analyses computing them. Thus, it can provide different analyses computing the same property kind to cover a wide range of precision, sound(i)ness, and performance trade-offs. Two experiments examine how this exchangeability fosters rapid probing, thus benefiting the analysis' developer and end user alike (**RQ2**): We explore the impact on precision in one experiment and that on soundness in the second.

In our first experiment, we run various configurations of our purity analysis (cf. Section 3.3) with different supporting analyses for field mutability or escape information with different precision-scalability trade-offs. No other tool supports similar exchangeability of interacting purity, mutability, and escape analyses. Table 2 shows the results for *hsqldb*. Higher indices indicate more precise analyses. Comparing the least precise analysis PA_0 with the most precise $PA_2/FMA_1/E_1$, we observe a reduction in the number of reported impure methods by ~28%, but a runtime slowdown by 18.6x. Some configurations even have a large impact on runtime for almost no gain in precision, e.g., comparing the most precise one with that using simpler escape analysis E_0.

In the second experiment, we evaluate the RTA call graph with different supporting modules for different JVM features. While DOOP computes call graphs and offers some modularity, e.g., for reflection, no other tool so far includes such fine-grained modules for

Table 3: Results for different call graph modules for Xalan

Configuration	#Reachable Methods	#Edges	⏱ Analysis
RTA	6 141	46 946	8.58 s
RTA_C	6 162	47 154	8.76 s
RTA_R	8 404	63 821	10.07 s
RTA_X	12 937	106 516	12.99 s
RTA_C_X	12 958	106 743	12.86 s
RTA_S_T_F_C_X	12 970	106 778	13.35 s

C=Configured native methods; R=Reflection; X=Tamiflex; S=Serialization; T=Threads; F=Finalizer;

call graphs. Also, DOOP does not support RTA, but points-to based call graphs only. Results for *Xalan* are shown in Table 3, displaying the active modules, the number of reachable methods (RMs), call edges, and respective construction time. While some configurations discover more methods/edges than others, they may discover different sets of methods/edges. A configuration is only guaranteed to be strictly more sound if it uses a strict superset of modules. Compared to the baseline, RTA with support for preconfigured native methods (RTA_C), reaches 21 more methods and ~200 more call edges. Reflection support (RTA_R) brings over 2 000 more RMs and 16 000 call edges; at the same time, construction time increases by about 15%. Using the Tamiflex (RTA_X) module instead increases call graph size (and soundness) more but introduces further slowdown. With all modules enabled, we reach 111% more methods and 127% more call edges, at the cost of a 55% increased runtime. Moreover, the data suggests that different modules benefit different projects. Tamiflex impacted *Xalan* and *jython*, reflection *fop*, and serialization *hsqldb*. Thus, which modules are more relevant than others may differ between different programs and it may be worth investigating tradeoffs even at the level of individual projects.

Overall, both experiments confirm that OPAL maintains exchangeability benefits from Datalog-based analyses, while generalizing these results to a broader range of lattices.

■ *OPAL facilitates systematic investigation of different configurations, supporting users and developers in finding the best trade-off between precision, sound(i)ness, and scalability.*

5.3 Parallelization

We implemented a proof-of-concept parallel version of our blackboard control (**RQ3**). Using this, we measured the execution time for the points-to-based call graph with different numbers of threads. Results for five DaCapo projects are shown in Figure 2. The projects were selected to have similar runtime to facilitate graph readability, the other projects show similar behavior. Benefits of parallelization over one thread appear at two to four threads and we achieve speedups of up to 2x for 16 threads. Beyond this, further improvement is negligible; instead, it slightly decreases due to growing communication overhead. These results are encouraging, given that the parallel version is not at all optimized. An optimized version of it is expected to scale better. Designing such an optimized version requires further research to identify the optimal way to parallelize the computation.

■ *OPAL's computation can be parallelized and that parallelization holds potential for increased performance.*

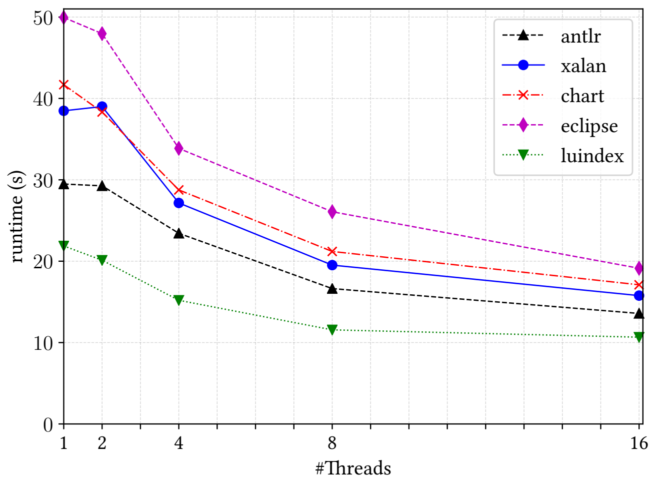

Figure 2: Parallel architecture performance

Table 4: Runtime and size of points-to based call graphs

	DOOP				OPAL		OPAL
Project	Compile	Facts	Analysis	#RM	runtime	#RM	(Scala)
antlr	107 s	35 s	41 s	8 402	28.36 s	8 653	305.90 s
bloat	109 s	21 s	33 s	9 644	34.43 s	10 000	266.08 s
chart	109 s	38 s	45 s	12 058	40.13 s	12 268	516.37 s
eclipse	109 s	19 s	17 s	7 163	44.89 s	13 429	343.69 s
fop	110 s	41 s	35 s	7 300	18.87 s	7 509	56.64 s
hsqldb	109 s	38 s	32 s	7 097	19.65 s	7 455	55.69 s
jython	108 s	24 s	90 s	12 901	77.65 s	13 161	3 341.62 s
luindex	108 s	21 s	19 s	7 608	19.34 s	7 972	62.57 s
lusearch	108 s	21 s	20 s	8 281	21.03 s	8 540	70.55 s
pmd	109 s	39 s	36 s	8 817	21.47 s	9 028	75.47 s
xalan	108 s	37 s	30 s	7 111	35.59 s	13 330	246.97 s
geo. ∅	108.54 s	29.09 s	32.51 s		29.68 s		191.26 s

5.4 Benefits of Specialized Data Structures

To answer **RQ4**, we compare two versions of the same points-to based call-graph algorithm. Both encode points-to, caller, and callee information as integer values. The first version uses specialized trie-based data structures, the second one uses standard Scala sets.

Results are given in the sixth and last column of Table 4. Due to its high memory consumption, we had to run the version using Scala's data structures with 128 GB of heap space; *jython*'s analysis even required 256 GB. Using tailored data structures, OPAL's runtime decreased by 65% to 98% compared to naively using Scala's sets.

■ *Selecting suitable data structures adapted to the specific analysis needs is an important factor for analysis performance. While the analysis developer can freely select optimized data structures in OPAL, strictly declarative approaches do not support such choices.*

5.5 Comparison with Declarative Approaches

After evaluating individual unique features of OPAL in isolation, we present the results of an experiment that directly compares the performance of OPAL with that of Doop [12] (**RQ5**) - a highly optimized state-of-the-art tool for declarative Java points-to and call-graph analyses on top of the Soufflé [45] Datalog engine. Its declarative approach assembles a fair comparison as it supports similar modularity and configurability and good trade-offs between pluggable precision/recall. Also, Doop's and Soufflé's authors repeatedly claimed its good performance [11, 12, 45, 71]. Specifically, we compare our points-to based call-graph's runtime from Section 3.2 to Doop's.

For better comparability, we disabled the reflection support in both tools, because the respective approaches are different. The applications were analyzed together with OpenJDK 1.7.0_75 (used for the TamiFlex data in Doop's benchmarks). Minor differences (less then 5% difference in the number of RMs, except for eclipse and xalan) remain, but these are in Doop's favor, since they result in more work to be done by OPAL[6]. Still, the sixth column of Table 4 shows that our complete analysis, including all preprocessing, is often faster than Doop's analysis (9% in the geometric mean). Further, Doop additionally requires time for rule compilation and fact generation.

We used OPAL's single-threaded implementation since it seems that Doop is hardly parallelized (fact generation was done with 128 threads, but did not significantly vary with other values for the `fact-gen-cores` parameter and the `souffle-jobs` parameter did not show any effects). Using a parallel version, OPAL should be able to outperform Doop even more as shown in Section 5.3.

■ *Despite being more general, i.e., not tuned for points-to analyses but supporting many different kinds of analyses, OPAL clearly outperforms Doop.*

6 RELATED WORK

In this section, we discuss several related approaches in various areas of static analysis as well as in blackboard systems.

6.1 Blackboard Systems

The blackboard metaphor was introduced by Newell [58] and implemented for speech-recognition in HEARSAY-II [31]. Blackboard systems were used for image recognition [54], vessel identification [61], or industrial process control [25]. For these domains, no efficient, deterministic algorithm is known, leading to several problems mentioned by Buschmann et al. [14]: nondeterminism making testing difficult, no guarantee for good solutions, performance suffering from wrong hypotheses, and high development effort due to ill-defined domains. As static analyses have a well-defined domain and deterministic algorithms, these do not apply to our approach.

The structure of blackboard systems is described, e.g., by Nii [60], Craig [21], and Corkill [17]. Corkill also discusses concurrent execution of knowledge sources and the control component [16], similar to OPAL. OPAL resembles a more modern interpretation of blackboard systems [22]: its blackboard is not hierarchical and analyses may keep state between activations. Information is, however, never erased and all communication is done via the blackboard.

Brogi and Ciancarini used the blackboard approach to provide concurrency for their Shared Prolog language [13]. Like static analyses, this domain is well-defined. Their knowledge sources are restricted to be Prolog logical programs, while OPAL's analyses can be implemented in a way best suited to the analysis needs.

Decker et al. [24] discuss the importance of heuristics for scheduling concurrent knowledge source activations. Focusing on static analyses and well-defined dependency relations, OPAL provides good general heuristics which are agnostic to individual analyses.

6.2 Abstract Interpretation

Cousot et al. [19] have proven that multiple (possibly cheap) abstract domains (i.e., analyses) can be combined using the reduced product to increase overall precision. In abstract interpreters, such as Astrée [20] or Clousot [32], dependencies between domains are restricted by the execution order. Thus, the same program statement must be analyzed multiple times which is superfluous with OPAL's explicit dependency management. Also, abstract interpretation typically aims to compute abstract *approximations* [18] of concrete values, such as an integer variable's value. OPAL further allows natural expression of analyses on all granularity levels. Keidel et al. [47, 48] provide modular and reusable abstract semantics for different language features allowing soundness proofs from composition of already sound components. The analyses again approximate single concrete program values. OPAL supports analyses to be based on abstract interpretation and includes such analyses, but generalizes to a much broader range of static analyses.

6.3 Declarative Analyses Using Datalog

Datalog is often used to implement static analyses in a strictly declarative fashion [27, 35, 51, 67, 77, 78]. Properties are represented as relations and rules specify how to compute them. This enables modularization, as rules can be easily exchanged and/or added (e.g. for new language features). The Doop [12] framework, building on top of the highly optimized Datalog solver Soufflé [45], has shown that the rule-based approach enables precise and scalable points-to analyses. For this reason, Doop became the state-of-the-art for such analyses [46, 70, 72, 74, 75]. Datalog-based frameworks, however, are limited in their expressiveness by using relations, i.e., set-based abstractions, to represent all analysis results. OPAL's approach combining imperative and declarative features provides similar benefits as Datalog-based approaches, while allowing for more expressive ways to represent data and to implement analyses.

Datalog's limitation to relations has also been pointed out by Madsen et al. [56]. They propose Flix to overcome this using a language inspired by Datalog and Scala to specify declarative pluggable analyses using arbitrary lattices as in OPAL. However, Flix focuses on verifying soundness and safety properties of static analyses and not on performance. For instance, Flix does not allow optimized data structures or scheduling strategies. We wanted to compare our approach against Flix and contacted the authors, but they answered that their IFDS implementation is dysfunctional now and suggested comparing against Doop with the Soufflé engine, which we did in Section 5.5. Szabó et al. [73] also extend Datalog to allow arbitrary lattices for static analysis. Their solver IncA focuses on incrementalization. OPAL allows optimizations, e.g., of used data structures or scheduling strategies. Furthermore, analyses' coarser granularity compared to individual rules reduces overhead in parallelization.

[6] For instance, OPAL does handle some cases of reflection more soundly even with reflection handling disabled in order to process the DaCapo benchmark correctly.

6.4 Attribute Grammars

Attribute grammars [49] used in compilers such as JastAdd [30] enable modular inference of program properties by adding computation rules to the nodes of a program's abstract syntax tree (AST). In traditional attribute grammars, attributes may only depend on parent, sibling, and child nodes. Circular reference attribute grammars [33, 37, 44, 57] enable attributes to depend on arbitrary AST nodes and allow circular dependencies. Still, analyses are tightly bound to the AST, impeding natural expression of analyses based on different structures, such as a control-flow graph. Similar to OPAL, JastAdd enables pluggability for new language features. However, JastAdd requires at least one attribute in a cyclic dependency to be marked explicitly, while OPAL handles this transparently.

Öqvist and Hedin [62] proposed concurrent evaluation of low complexity attributes in circular reference attribute grammars. OPAL on the other hand supports arbitrary granularity of concurrent computation. OPAL's explicit dependency management enables analyses to drop dependencies and commit final results early for improved performance. Finally, as memorization of properties is done in OPAL's blackboard, temporary values are garbage collected automatically, whereas JastAdd requires explicit removal.

6.5 Imperative Approaches and Parallelization

Lerner et al. [52] proposed modularly composed dataflow analyses which communicate implicitly through optimizations of the analyzed code or explicitly through *snooping*. A fixed-point algorithm repeatedly reanalyzes the code, while OPAL's explicit dependencies avoid reanalysis. They support only dataflow analyses, while OPAL enables a wide range of analyses including dataflow analyses.

CPAchecker [5] is a tool for configurable software verification and analysis. For any combination of analyses, CPAchecker requires defining a compound analysis to integrate results of individual analyses and manage their interaction. For CPA+ [6], combined analyses must work with the same domain and provide an explicit measure of result precision. In contrast, OPAL enables tight interaction and interleaved execution of independently-developed analyses without requiring a compound analysis or explicit measure of precision.

Johnson et al. [43] present a framework for collaborative alias analysis. Clients ask queries which are processed by a sequence of analyses. Each analysis can answer the query or forward it to the next one. Analyses can also generate additional (premise) queries. To ensure termination, a complexity metric must be defined and premises must be simpler than the queries they originate from. Therefore, cyclic dependencies, required for optimal precision, and results combined from different analyses are not supported.

Parallel execution of static analyses is performed by Magellan [29]. In this framework, dependencies are given by the data processed instead of explicitly by the analyses.

Haller et al. [36] concurrently execute tasks based on lattices and apply this to static analysis. Their framework requires dependencies to be managed fully by the client while OPAL manages them automatically based on declarative specifications. In recent work [39], we extended this approach to support mutable state and found that exchangeable scheduling strategies significantly impact performance. Both concepts are supported in OPAL.

7 THREATS TO VALIDITY

One threat to the validity of our evaluation is the use of the relatively old and small DaCapo benchmark. It is, however, widely used to evaluate Doop [12] and to compare other approaches with Doop [2, 3, 63, 74]. Doop's special support for the benchmark makes it a particularly fair evaluation set. Furthermore, our experiment design, based on relative comparisons, should yield the same results with any well-assembled benchmark.

Also, our results are threatened if our points-to analysis is not sufficiently similar to Doop. To achieve comparability, we tailored our points-to analysis to be as similar as possible, i.e., the call graph derived from the points-to results should be almost identical. In order to ensure this, we systematically studied Doop's Datalog rules, validated the resulting call graphs using Judge [65] and manually inspected points-to sets from deviating call graphs.

8 CONCLUSION

We presented a novel approach for modular collaborative static analyses implemented in the OPAL framework. Like with declarative frameworks such as Doop, OPAL's analyses, while developed in isolation, can be easily composed to complex analyses by collaboratively computing results during interleaved executions. Subanalyses can be reused in various complex analyses and one can easily exchange sub-analyses of a complex analysis for fine-tuning precision, sound(i)ness, and performance.

But, instead of relying on a general-purpose solver, OPAL combines imperative and declarative features to overcome limitations of fully declarative frameworks. Individual analyses can be implemented in imperative style making use of whatever data structures and implementation strategies are appropriate for their specific needs. Interdependencies and other characteristics important for guiding their interleaved execution are declaratively specified and automatically managed by a custom solver resembling a blackboard architecture. Due to its approach, OPAL (a) is more general in terms of the analyses supported - it is in particular the first framework to explicitly support lazy collaboration of optimistic and pessimistic analyses - and (b) enables analysis-specific optimizations, which lead to outperforming state-of-the-art declarative analyses.

We plan to explore several further research questions in the future. First, our evaluation suggests that better parallelization strategies for OPAL are needed. Second, we plan to explore further scheduling strategies, both general and analysis specific, e.g., strategies that abort computations whose results are no longer of interest, or strategies (as well as analyses) that adapt their behavior during the execution to increase performance with minimal impact on precision and/or soundness. Last but not least, we will develop a formal model of OPAL and formally prove its properties. For instance, we believe that OPAL's design enables compositional soundness proofs [47, 48] - this needs to be proved in the future.

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] Alexander Aiken. 1999. Introduction to set constraint-based program analysis. *Science of Computer Programming* 35, 2-3 (1999), 79–111.

[2] Karim Ali and Ondřej Lhoták. 2012. Application-only call graph construction. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 688–712.

[3] Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 378–400.

[4] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. *ACM Sigplan Notices* 31, 10 (1996), 324–341.

[5] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. 2007. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*. Springer, 504–518.

[6] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. 2008. Program analysis with dynamic precision adjustment. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 29–38.

[7] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 169–190.

[8] Eric Bodden. 2012. Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis (SOAP)*. 3–8.

[9] Eric Bodden. 2018. The Secret Sauce in Efficient and Precise Static Analysis. In *International Workshop on State Of the Art in Java Program analysis,(SOAP)*. 84–92.

[10] 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 (ICSE)*. ACM, 241–250.

[11] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception analysis and points-to analysis: better together. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA)*. ACM, 1–12.

[12] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses *(OOPSLA)*.

[13] Antonio Brogi and Paolo Ciancarini. 1991. The concurrent language, shared prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 99–123.

[14] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *A System of Patterns*. Pattern-Oriented Software Architecture, Vol. 1. Wiley.

[15] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *OOPSLA*.

[16] Daniel D Corkill. 1989. Design alternatives for parallel and distributed blackboard systems. In *Blackboard Architectures and Applications*.

[17] Daniel D Corkill. 1991. Blackboard systems. *AI expert* 6, 9 (1991), 40–47.

[18] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*. ACM, 238–252.

[19] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*. ACM, 269–282.

[20] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of abstractions in the ASTRÉE static analyzer. In *Annual Asian Computing Science Conference (ASIAN)*. Springer, 272–300.

[21] Iain D Craig. 1988. Blackboard systems. *Artificial Intelligence Review* 2, 2 (1988), 103–118.

[22] Iain D Craig. 1993. *A New Interpretation of The Blackboard Metaphor*. Technical Report. Technical report, Department of Computer Science University of Warwick.

[23] 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 (ECOOP)*. Springer, 77–101.

[24] Keith Decker, Alan Garvey, Marty Humphrey, and Victor R Lesser. 1991. Effects of Parallelism on Blackboard System Scheduling. In *International Joint Conferences on Artificial Intelligence (IJCAI)*. 15–21.

[25] Roland Dodd, Andrew Chiou, Xinghuo Yu, and Ross Broadfoot. 2009. Industrial process model integration using a blackboard model within a pan stage decision support system. In *2009 Third International Conference on Network and System Security (NSS)*. IEEE, 489–494.

[26] Michael Eichberg and Ben Hermann. 2014. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, ACM, 1–6.

[27] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. 2008. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering (ICSE)*. ACM, 391–400.

[28] 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*. ACM, 113–118.

[29] Michael Eichberg, Mira Mezini, Sven Kloppenburg, Klaus Ostermann, and Benjamin Rank. 2006. Integrating and Scheduling an Open Set of Static Analyses. *(ASE)*.

[30] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd Extensible Java Compiler. In *Proceedings of the ACM on Programming Languages (OOPSLA)*.

[31] Lee D Erman, Frederick Hayes-Roth, Victor R Lesser, and D Raj Reddy. 1980. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys (CSUR)* 12, 2 (1980), 213–253.

[32] Manuel Fähndrich and Francesco Logozzo. 2010. Clousot: Static contract checking with abstract interpretation. In *International Conference on Formal Verification of Object-oriented Software (FoVeOOS)*. Springer, 10–30.

[33] Rodney Farrow. 1986. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *ACM SIGPLAN Notices*, Vol. 21. ACM, 85–98.

[34] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 685–746.

[35] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. 2006. Codequest: Scalable source code queries with datalog. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2–27.

[36] Philipp Haller, Simon Geries, Michael Eichberg, and Guido Salvaneschi. 2016. Reactive Async: expressive deterministic concurrency *(SCALA)*.

[37] Görel Hedin. 2000. Reference attributed grammars. *Informatica (Slovenia)* 24, 3 (2000), 301–317.

[38] Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. 2018. A unified lattice model and framework for purity analyses *(ASE)*.

[39] Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini. 2020. A Programming Model for Semi-Implicit Parallelization of Static Analyses. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[40] Wei Huang, Ana Milanova, Werner Dietl, and Michael D Ernst. 2012. Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*. 879–896.

[41] IBM. [n.d.]. WALA - Static Analysis Framework for Java. http://wala.sourceforge.net/. [Online; accessed 12-June-2020].

[42] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2010. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*. Springer, 320–339.

[43] Nick P Johnson, Jordan Fix, Stephen R Beard, Taewook Oh, Thomas B Jablin, and David I August. 2017. A collaborative dependence analysis framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO)*. IEEE Press, 148–159.

[44] Larry G Jones. 1990. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 429–462.

[45] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification (CAV)*. Springer, 422–430.

[46] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 423–434.

[47] Sven Keidel and Sebastian Erdweg. 2019. Sound and Reusable Components for Abstract Interpretation. In *Proceedings of the ACM on Programming Languages (OOPSLA, Vol. 3)*. ACM, 176.

[48] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional soundness proofs of abstract interpreters. In *Proceedings of the ACM on Programming Languages (ICFP, Vol. 2)*. ACM, 72.

[49] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.

[50] Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (VEE)*.

[51] Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 1–12.

[52] Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing dataflow analyses and transformations *(POPL)*.

[53] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*. Springer, 153–169.

[54] Hongyi Li, Rudi Deklerck, Bernard De Cuyper, A Hermanus, Edgard Nyssen, and Jan Cornelis. 1995. Object recognition in brain CT-scans: knowledge-based fusion of data from multiple feature extractors. *IEEE Transactions on medical imaging (TMI)* 14, 2 (1995), 212–229.

[55] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[56] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to FLIX: A declarative language for fixed points on lattices. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 194–208.

[57] Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars—their evaluation and applications. *Science of Computer Programming* 68, 1 (2007), 21–37.

[58] Allen Newell. 1962. *Some problems of basic organization in problem-solving programs*. Technical Report. Rand Corp Santa Monica CA.

[59] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2005. *Principles of Program Analysis*.

[60] H Penny Nii. 1986. The blackboard model of problem solving and the evolution of blackboard architectures. *AI magazine* 7, 2 (1986), 38–38.

[61] H Penny Nii, Edward A Feigenbaum, and John J Anton. 1982. Signal-to-symbol transformation: HASP/SIAP case study. *AI magazine* 3, 2 (1982), 23–23.

[62] Jesper Öqvist and Görel Hedin. 2017. Concurrent circular reference attribute grammars. In *10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. ACM, 151–162.

[63] Edgar Pek and P Madhusudan. 2014. Explicit and symbolic techniques for fast and scalable points-to analysis. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP)*. ACM, 1–6.

[64] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. 2000. Automatic detection of immutable fields in Java. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*. IBM Press, 10.

[65] 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 (ISSTA)*. ACM, 251–261.

[66] 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*. 2–7.

[67] Thomas W Reps. 1995. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*. Springer, 163–196.

[68] Thomas W Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM, 49–61.

[69] Jonathan Rodriguez and Ondřej Lhoták. 2011. Actor-based parallel dataflow analysis *(CC)*.

[70] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *Asian Symposium on Programming Languages and Systems*. Springer, 485–503.

[71] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop*. Springer, 245–251.

[72] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 17–30.

[73] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[74] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*. Springer, 489–510.

[75] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. *ACM SIGPLAN Notices* 52, 6 (2017), 278–291.

[76] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[77] John Whaley. 2007. *Context-sensitive pointer analysis using binary decision diagrams*. Ph.D. Dissertation. Stanford University.

[78] John Whaley and Monica S Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *ACM SIGPLAN Notices* 39, 6 (2004), 131–144.