A Unified Lattice Model and Framework for Purity Analyses

Dominik Helm
TU Darmstadt
Germany
helm@cs.tu-darmstadt.de

Florian Kübler
TU Darmstadt
Germany
kuebler@cs.tu-darmstadt.de

Michael Eichberg
TU Darmstadt
Germany
eichberg@cs.tu-darmstadt.de

Michael Reif
TU Darmstadt
Germany
reif@cs.tu-darmstadt.de

Mira Mezini
TU Darmstadt
Germany
mezini@cs.tu-darmstadt.de

ABSTRACT

Analyzing methods in object-oriented programs whether they are side-effect free and also deterministic, i.e., *mathematically pure*, has been the target of extensive research. Identifying such methods helps to find code smells and security related issues, and also helps analyses detecting concurrency bugs. *Pure* methods are also used by formal verification approaches as the foundations for specifications and proving the *pureness* is necessary to ensure correct specifications.

However, so far no common terminology exists which describes the purity of methods. Furthermore, some terms (e.g., *pure* or *side-effect free*) are also used inconsistently. Further, all current approaches only report selected purity information making them only suitable for a smaller subset of the potential use cases.

In this paper, we present a fine-grained unified lattice model which puts the purity levels found in the literature into relation and which adds a new level that generalizes existing definitions. We have also implemented a scalable, modularized purity analysis which produces significantly more precise results for real-world programs than the best-performing related work. The analysis shows that all defined levels are found in real-world projects.

CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Software and its engineering → Automated static analysis**;

KEYWORDS

Purity, side-effects, static analysis, lattice, Java.

ACM Reference Format:
Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. 2018. A Unified Lattice Model and Framework for Purity Analyses. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238226

1 INTRODUCTION

Static analyses that identify side-effect free and also deterministic, i.e., *mathematically pure*, methods in object-oriented programs have been the target of extensive research [25, 35, 37, 38]. Identifying such methods helps to improve analyses detecting concurrency bugs [20] and to find security-related issues [20, 37]. Pure methods written in programming languages such as Java are also used by formal verification approaches as the foundation for the respective specifications [6, 15, 16]. In that case, it is necessary to prove a method's purity to ensure that the formal specifications are correct. The identification of (mostly) pure methods further facilitates program comprehension [7, 18], provides opportunities for code optimizations [11, 28, 43], and supports testing as well as automated verification [41]. Recent trends towards a more functional style of programming relying on pure methods [1], also demonstrate the overall relevance.

However, no common terminology exists which describes the purity of methods [34] and some terms (e.g., *pure* or *side-effect free*) are also used inconsistently. Further, all current approaches only report selected purity information and are thus only suitable for a small subset of potential use cases. Identifying side-effect free and deterministic methods enables compilers to perform compile time evaluations. Formal verification approaches need the information to ensure correctness. Detecting concurrency bugs and code smells requires *just* side-effect free methods while code comprehension can benefit from all levels and also weakened forms of purity.

In this paper, we present a fine-grained unified lattice model for specifying a method's purity. In the model, each of the 13 lattice elements has a well-defined semantics and is put into relation to the purity levels found in the literature. The model is extended by the level (*Contextual Purity*) which generalizes the so-called *External Purity* [7]. Being able to ignore specific operations in specific contexts [37], e.g., logging in business applications, is also supported and generalized to *Domain-specific Purity.* The proposed model is sufficiently detailed for all identified use cases. Furthermore, the purity levels (*External* and *Contextual* Purity) support purity analyses to rate methods as pure if the called methods have side-effects that are limited to the caller. Therefore, the lattice model is also a suitable target for modular purity analyses that reason about each method in isolation. Additionally, we present a scalable, purity analysis that produces more precise results (> 4%) for real-world code than the state-of-the-art. The analysis infers purity for individual methods in isolation and relies on the results of several independent analyses. Using the analysis, we will show that all defined levels are relevant when analyzing real-world projects.

In Section 2 we will discuss the state-of-the-art. After that, we will present the *Unified Lattice Model* in Section 3. The implementation of the proposed analysis (*OPIUM*[1]) is discussed in Section 4 and evaluated in Section 5. Section 6 concludes the paper.

2 RELATED WORK

In the following, we discuss previous research on purity. We will present the purity definitions found in the literature alongside with the proposed analyses. We will start with those approaches that focus on identifying side-effect free methods – independent of the question whether the methods are also deterministic or not. After that, we will discuss those approaches that rely on stricter *purity* definitions [5, 13, 23, 25, 26, 32, 33, 38, 39].

2.1 Side-Effect Free Methods

One of the most commonly used definitions of method purity is given by Sălcianu and Rinard [38, 39] where a method is pure if "*it does not mutate any object that exists in the pre-state,* i.e., *the program state right before the method invocation*". This definition does not capture deterministic behavior, but it ensures the absence of side-effects. Their analysis is combined with a pointer analysis ([24]) and also supports the identification of individual parameters that are *read-only* or *safe*. Read-only parameters are parameters of a method where no object transitively reachable through this parameter is mutated by the method. Safe parameters additionally require that no new externally visible paths to objects reachable through them are created. To classify the parameters, their analysis is able to determine which memory locations may be modified by a method. The analysis is used in the Korat [8, 31] tool to check the purity of methods specifying the behavior of data structures.

The same definition of purity is used by Huang et al. [25, 26]. They propose to extend the Java type system and present a type inference algorithm to annotate references as *read-only*. In the type system, pure methods are those that do not modify global state through static fields and that do not have any parameters inferred as mutable. The authors suggest to use the approach to find errors or to do optimizations in concurrent programs.

Pearce [33] uses a similar definition. His JPure system checks that methods annotated as *pure* do not modify previously existing program state. It is also capable of inferring purity annotations for code that is not annotated.

Genaim and Spoto [23] again refer to a method as pure, if it does not modify the heap structures reachable from any of its parameters. Their constancy analysis identifies the parameters which are not used to modify the reachable heap. The analysis uses alias relationships between the parameters expressed as boolean formulas.

In the approach proposed by Ierusalimschy and Rodriguez [27] side-effect free methods are allowed to allocate new objects and return them as long as the pre-state is not modified. They rely on manual annotations that mark methods as side-effect free. Their goal is to extend the type system and to automatically check the respective methods for conformance at compile-time.

[1]OPAL Purity Inference based on a Unified lattice Model.

In contrast, a dynamic analysis to find *pure* methods is described by Dallmeier [13]. The analysis explicitly deals with multithreading and especially the fact that constructors, while they may assign to fields of the currently initialized object, can be pure. His analysis results are used for ADABU, a tool for mining object behavior that requires information about side-effect free methods for classifying methods as observers and mutators [14].

The definition of side-effect free methods used by Rountev [35] is stricter than the previous ones. While it does allow allocation of new objects, these may not escape to the caller. The proposed model assumes single-threaded execution as the pre-state of the method could be modified by concurrently executing methods otherwise. They describe two analyses based on RTA ([3]) and a points-to analysis to identify side-effect free methods in partial programs.

Naumann [32] and Barnett et al. [5, 6] introduce the idea of *observational purity*. Such methods are allowed to have side-effects that are not observable by their callers. This definition especially allows for caching (intermediate) results, as is done in memoization. It is only valid in languages without unrestricted pointer arithmetic where noninterference properties can be proven [4]. Methods that are *observationally pure* could be used in program specifications written in, e.g., ESC/Java [21] and JML [29]. Traditionally, these languages required stronger restrictions (no use of methods in ESC/Java at all and only provably pure methods in JML). The analysis they propose to determine observational purity is built upon an information-flow analysis [36].

ESC/Java2 [12] uses side-effect free methods for specifications, but relies on programmer specified annotations to identify them. The authors also recognize that determinism is required for specifications, but do not provide a way of identifying methods that are deterministic and side-effect free.

Table 1 summarizes the different approaches to detecting side-effect free methods and the terminology used by the authors.

2.2 Deterministic Purity

Aside from not performing side-effects, deterministic behavior (i.e., producing the same outputs whenever invoked with the same parameters) is a necessary condition for methods to be referentially transparent. This is required for compiler optimizations as well as formal specifications.

The term *functionally pure* for methods that are deterministic and side-effect free is introduced by Finifter et al. [20]. They use a subset of the Java language called Joe-E that restricts some features of Java that are non-deterministic or cause side-effects, including, mutable static state and access to the stack trace of exceptions. Pure methods are automatically thread-safe, as they can never interfere with other threads, and no synchronization is required. This allows for verifying security properties such as the correct behavior of encoding and decoding methods. They also suggest using side-effect free methods for assertions and specifications. Similar to the concept of Huang et al. [26], methods that only have immutable parameters can never cause side-effects or be non-deterministic, so pure methods can be easily identified.

In their work on dynamic purity analysis, Xu et al. [42] define several levels of purity. *Strongly pure* methods must be side-effect

Table 1: Summary of analyses for side-effect free methods

Authors	Analysis type	Purity levels (as named by authors)
Sălcianu/Rinard [38, 39]	pointer analysis	pure
Huang et al. [25, 26]	type system extension	pure
Pearce [33]	annotations	pure
Genaim/Spoto [23]	parameter mutability	pure
Ierusalimschy/Rodriguez [27]	type system extension	side-effect free
Dallmeier [13]	dynamic purity analysis	pure
Rountev [35]	static purity analysis	side-effect free
Barnett et al. [6]	information flow analysis	strongly pure, observationally pure

Table 2: Summary of analyses for deterministic pure methods

Authors	Analysis type	Purity levels (as named by authors)
Finifter et al. [20]	type system/restricted language	side-effect free, functionally pure
Xu et al. [42]	static & dynamic purity analysis	strong, moderate, weak, once-impure
Zhao et al. [43]	static purity analysis	pure
Benton/Fischer [7]	type-and-effect system	pure, read-only, externally pure, externally read-only
Stewart et al. [37]	type system extension	strict, strong, weak, externally pure

free and deterministic and are only allowed to have primitive parameters, thereby excluding reference type parameters completely. They may also not create any new objects or call impure methods, even if the effects of both are not visible to the caller. *Moderately pure* methods are similar in the constraints on their inputs, but may create new objects as long as they don't escape the method execution context, similar to the definition of Rountev [35] above. They may also call impure methods if their effects are not observable by the caller. The restriction on reference types is partially lifted in *weakly pure* methods that may access fields of object type parameters. A rather unique concept is *once-impure purity* that allows methods to be impure on their first, but not on subsequent invocations. The authors do not detail the uses cases, but it seems that they want to support lazy initialization patterns. While their work focuses on dynamic analysis, they also present a static analysis for strong purity. The analysis divides the bytecode instructions executed by a method into impure and pure instructions. For weaker purity levels, some instructions are considered pure only when they are performed on locally allocated, non-escaping objects. The analysis results are used to support automated memorization of method results. This is possible because the results of pure methods are the same when invoked again with the same parameters.

Zhao et al. [43] explore different approaches to find pure methods: automated checking of programmer supplied annotations and two static analyses based on a method's bytecode. The purity information is then used inside the Jikes VM [2] to support further analyses and optimizations such as the elision of method calls. Unnecessary synchronization can be removed as pure methods do not require synchronization with another thread.

External purity is introduced by Benton and Fischer [7]. Externally pure methods are allowed to read and modify mutable state, but only on the receiver object of a call. Constructors that leak the reference to currently initialized objects are ignored as the authors consider this to be rare. The weaker purity level *externally read-only* allows methods to modify the state of the receiver object as above and to read any mutable state. They show that a large percentage of methods in object-oriented programs fulfills these conditions using a type-and-effect system [30].

Stewart et al. [37] extend ReImInfer [25] by combining previous definitions of purity into five levels of side-effects. For two of the proposed levels *strict purity* (no local variable assignments are allowed) and *strong purity* (no allocation of objects are allowed) no use cases were identified and – as the authors admit – both are of no practical relevance. They also discuss three properties related to *Input*, *Output* and *Determinism*, but treat them as orthogonal to the purity levels. A coherent lattice model is not defined. Lastly, the tool is not available rendering an empirical evaluation of it impossible.

A summary of approaches identifying deterministic pure methods and the terminology used by the authors is given in Table 2.

3 MODEL

In the following, we discuss the unified lattice model by defining the different purity levels and their relations as well as by comparing them to the levels defined in the literature. The presented model is generic and can be used for any object-oriented programming language. Examples in the following sections are in Java.

3.1 Purity Levels

We first introduce *side-effect free* as it builds the foundation for other levels. We will then proceed to discuss purity, before we finally present weaker purity levels.

3.1.1 *Side-effect Free Methods.*

Definition 1. *A method is* side-effect free *if all object-graph manipulations, which are performed by the method or its callees, are only visible to the method while it is executed,* i.e., *all manipulations are invisible to a method's client.*

Here, the object graph is considered to also include the system's resources, e.g. the file system, network etc. and therefore methods manipulating these resources are not *side-effect free*.

This definition refines definitions found in previous work which are based on pre-state manipulations. In a multi-threaded environment, a method invocations' exact pre-state and whether changes to objects allocated on concurrent threads constitute a side-effect, is unclear. A particularly problematic definition is that of Rountev [35], where the object graph must be equivalent before and after the method invocation. With concurrently executing threads, even a temporary change of the object graph can affect the results of the program. The proposed definition can be applied in multi-threaded environments, as it explicitly refers to the visibility of side-effects, *i.e.,* it does not matter whether the method's pre-state is modified or not.

This definition also allows *side-effect free* methods to return newly allocated objects, as the modification of the object graph only become visible to the direct caller after the method execution.

Side-effect free methods may invoke methods that are not *side-effect free* themselves, iff the caused side-effects are confined to the calling method, *i.e.,* the side-effect is invisible to callers of the *side-effect free* method. An example is shown in the following Listing 1:

```
class CounterValue {
  public static int counter;
  private int value;
  public static CounterValue getCurrentValue(){
    CounterValue current = new CounterValue();
    current.setToCounter();
    return current;
  }
  private void setToCounter(){
    value = counter;
  }
}
```

Listing 1: A side-effect free method.

As the method `setToCounter` modifies its receiver object, it has a side-effect. However, when `setToCounter` is invoked by `getCurrentValue`, this side-effect is confined to the method's scope, i.e., the side-effect occurs on a newly allocated object that is invisible to other methods than `getCurrentValue` until it finishes execution. Thus, `getCurrentValue` is *side-effect free*.

`java.lang.System.currentMillis()` is a Java example of a *side-effect free* method.

Further, methods that perform synchronization – except locking of objects inaccessible by other threads – are not *side-effect free* as they change the monitor's state causing changes to an object's object graph. Hence, concurrently executed *side-effect free* methods can not cause deadlocks, livelocks, or race conditions since they do not modify any state that is visible outside their execution scope.

Side-effect freeness is the foundation of further purity definitions and many authors refer to *side-effect free* methods as *pure* (e.g. [26, 33, 38]). In order not to confuse side-effect freeness with functional purity – that also requires deterministic behavior –, we will refer to non-deterministic methods without side-effects as *side-effect free* as in [27].

3.1.2 *Pure Methods.*

Definition 2. *A method is* pure *if it is* side-effect free *and additionally produces a structurally equal, deterministic result each time the method is invoked with identical parameters during one execution of a program. Two reference parameters are identical if they reference the same object. The results are structurally equal if the returned object graphs are isomorphic; ignoring aliasing relationships. Primitive parameters are identical and structurally equal if both values are the same.*

This definition is similar to that of *weak purity* as described by [37] or [42] and ensures referential transparency, i.e. the possibility of replacing the method's invocation by its result.

```
final static double PI = 3.1415;
static double getArea(double radius){ return radius*radius*PI; }
```

Listing 2: A pure method.

The method `getArea`, shown in Listing 2, is *pure* by our definition. It deterministically computes the circle's area with the given radius and is free from side-effects.

Deterministic behavior can be achieved when neither mutable global state is used nor non-deterministic methods are called. Please note that determinism is not influenced by immutable global state like the mathematical constant `PI` in Listing 2 and, therefore, use of immutable global state is allowed.

In multi-threaded programs, fields of reference-type parameters may change during the execution of the method due to concurrent threads. We therefore restrict accesses to immutable fields. Others have opted for more restrictive approaches, *e.g.,* Xu et al. only allow value-type parameters for their definitions of *strongly* and *moderately* pure methods [42]. Alternatively, only immutable reference-type parameters can be allowed [20].

We do not apply special treatment to data that is immutable during the execution of a program, but may be initialized differently in another program execution. *Pure* methods may use such data and may return different results across different program executions. As long as the results are identical during a single program execution, those methods are considered *pure*.

Due to the existence of aliasing, it is necessary to define when parameters of different invocations are considered equal. We require *pure* methods to return the same result when objects with the same identity are passed as parameters, *i.e.,* objects at the same memory location. This restriction eases the implementation, as it allows to treat comparisons for reference (in)equality as deterministic. As mutable state from objects may not be accessed by *pure* methods, it is not necessary that all objects reachable via a parameter or primitive fields of a parameter object are identical too. If the parameter object itself has the same identity, different field values can only arise for mutable fields that can not be accessed by *pure* methods anyway. For a method's result, we require structurally equal object graphs, *i.e.,* isomorphic graphs without respect of aliasing relationships. Therefore, methods are allowed to return a newly allocated object for every execution. A potentially counterintuitive consequence is that it is unsound to reuse a *pure* method's result instead of reevaluating it when the result may be subject to a reference equality test as in Listing 3:

```
Object a = pureMethod();
Object b = pureMethod();
if(a == b) [...]
```

Listing 3: A problematic reference equality check.

Without further analysis, this may neither be reduced to a single call, nor may the calls be compile-time evaluated to a single or two distinct result objects.

One important limitation of this definition is that *pure* methods can only return those mutable objects for which the object graph is guaranteed to be structurally equal across method calls with identical parameters. For example, `getArray`, depicted in Listing 4, is not *pure*, as the content of the array may change in between method invocations. While it is difficult to prove structural equality of the object graph in general, many practical cases are easy to find, e.g. allocation and deterministic initialization of a mutable object will yield deterministic results if the object does not escape the method's execution context before being returned.

```
final static int[] array = new int[]{ 1 };
static int[] getArray(){
  return array;
}
```

Listing 4: A method that is not pure.

3.1.3 External Purity.

Definition 3. *A method is* externally pure *(*externally side-effect free*) if its invocation may lead to a modification of its receiver, but is* pure *(*side-effect free*) otherwise.*

Externally pure methods can be used to identify additional methods as either *pure* or *side-effect free* when callers of respective methods know, that the receiver object is confined to their context, *i.e.,* the receiver of the *externally pure* method does not escape the methods scope. Furthermore, a client analysis can use this purity level to trivially identify methods that break abstractions boundaries by, e.g., mutating global state. As recognized by Benton and Fischer [7], it is beneficial to identify such methods to improve program comprehension.

```
public class A {
  public int f;
  public void setField(int value){
    f = value;
  }
}
```

Listing 5: A field's setter that is externally pure.

Listing 5 gives an example of an *externally pure* method. It is deterministic and its only side-effect is the write to the field `f` which belongs to the same receiver object as the method.

Finding methods that are *externally pure* is essentially a specialized form of side-effect analysis.

External purity also applies to methods that use synchronization on the receiver object. This includes methods that have the `synchronized` modifier as well as methods with explicit synchronization on the receiver object. This follows directly from the above definition, as the monitor, that is used to perform the synchronization, is modeled as a property of the object.

In contrast to Benton and Fischer, we treat methods reading the receiver's mutable state as *side-effect free* instead of *externally pure.* This enables us to classify methods with calls on non-confined receiver objects as *side-effect free* rather than *impure.* The drawback is that a caller that invokes such methods on a confined receiver object can not be *pure* anymore.

3.1.4 Contextually Pure.

Definition 4. *A method is* contextually pure *(*contextually side-effect free*) if its invocation may lead to a modification of at least one of its parameters, but is* pure *(*side-effect free*) otherwise.*

We define *contextual purity* as an extension of *external purity.* It captures methods that potentially modify any of their parameters instead of only their receiver. While *contextually pure* methods break abstraction boundaries, they are still less problematic than methods with side-effects on (static) global state. Also, they allow identifying more side-effects as confined if it is known that none of their actual parameters are visible outside of their caller.

```
public class A {
  public int f;
  public static void modifyA(A a, int value){
    a.f = value;
  }
}
```

Listing 6: A contextually pure field setter

Consider `modifyA` in Listing 6. Its only side-effect is to modify parameter `a`'s state deterministically. Thus, it is *contextually pure.* An example of a very frequently used *contextually pure* Java method is `System.arraycopy()`. It modifies the given target array and is used in implementations of many core data structures.

3.1.5 Domain-specific Purity.

Definition 5. *A method is* domain-specific pure *(*domain-specific side-effect free*) if it is* pure *(*side-effect free*) when the effects of certain instructions – belonging to a certain domain – are ignored. Here, two parameterized instructions with different parameterizations are considered different instructions, e.g., two field access instructions on different fields.*

Sometimes, methods perform side-effect causing operations that neither can be classified *pure* nor *side-effect free* by using the proposed definitions. However, when having a closer look, those operations neither influence the method's result nor lead to effects on other methods. For instance, a method that writes a log file admittedly causes a side-effect but this kind of side effect can be tolerated because it has no direct influence on any other method. This property kind was previously identified by Steward et al. [37].

```
static final double PI = 3.1415;
static double getArea(double radius){
  System.out.println("called getArea");
  return radius*radius*PI;
}
```

Listing 7: A domain-specific pure method performing logging.

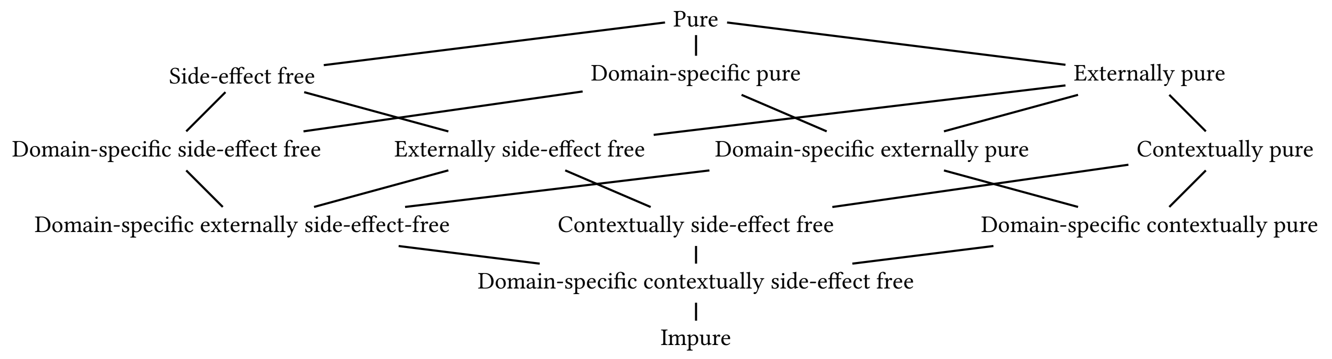
Pure
Side-effect free
Domain-specific pure
Externally pure
Domain-specific side-effect free
Externally side-effect free
Domain-specific externally pure
Contextually pure
Domain-specific externally side-effect-free
Contextually side-effect free
Domain-specific contextually pure
Domain-specific contextually side-effect free
Impure

Figure 1: The unified lattice for purity information

Consider the method getArea in Listing 7: Usually, the program's execution will not depend on the output from the println statement. Thus, the method can be considered as *domain-specific pure.* The possibility of classifying logging as pure is important for the analysis of enterprise applications that include logging in many methods.

Another example of domain-specific behavior is raising exceptions. This is not pure, as newly constructed exceptions contain the current stack trace and that is not deterministic w.r.t. parameters of the method invocation. For example, two method invocations with identical parameters may not lead to exceptions with the same stack trace. However, as the stack trace is usually not inspected by the program, treating it as *domain-specific pure* allows to classify further methods as effectively deterministic.

In contrast to previous work (e.g. [42]) which only treats explicit exceptions and ignores implicit ones (e.g. NullPointerExceptions raised by the JVM), our handling allows to consistently treat implicitly and explicitly thrown exceptions.

3.1.6 Orthogonal Purity Properties. The previously defined purity levels capture four different properties: (1) deterministic and non-deterministic behavior (*pure* vs. *side-effect free*), (2) modification of the receiver object (*external purity*), (3) modification of formal method parameters (*contextual purity*), and (4) non-deterministic or impure actions that may be considered pure in some circumstances (*domain-specific purity*). These properties are orthogonal to each other, *i.e.,* every property combination is possible except for *external* and *contextual purity*, as the latter subsumes the first.

We define the combinations of *externally pure* and *contextual pure* with *domain-specific pure* to get the benefits of *external* and *contextual purity.* This allows to apply the concept of *domain-specific purity* to methods that modify their receiver or parameters.

Definition 6. *A method is* domain-specific externally pure *(*domain-specific externally side-effect free*) if it is* externally pure *(*externally side-effect free*) and ignores the effects of specific instructions.*

Definition 7. *A method is* domain-specific contextually pure *(*domain-specific contextually side-effect free*) if it is* contextually pure *(*contextually side-effect free*) and ignores the effects of specific instructions.*

3.1.7 Impurity. When a method does not have any of the previously described properties and, therefore, no previous purity level can be assigned, we refer to it as *Impure*.

3.2 Purity Lattice

We arrange the purity levels defined above into a single, unified lattice that captures their relationships. The lattice enables a monotone framework for increasingly precise purity analyses that are able to refine previous analyses' results.

The purity lattice is depicted in Figure 1. Its top element is *pure*, the strictest purity level. Each step down the lattice loosens exactly one restriction on the method: *Side-effect free* allows for non-deterministic behavior, *domain-specific* allows domain-specific side-effects like logging, and *externally* allows modifications on the implicit this parameter. *Contextually* loosens further restrictions, allowing modification all formal method parameter (including this). *Impure* represents bottom value that places no restrictions on the method.

4 PURITY ANALYSIS

The purity analysis' implementation is based on the fix-point computations framework (FPCF) provided by OPAL [19]. The framework supports a decoupled implementation of mutually dependent analyses and automatically resolves cyclic dependencies in a sound manner. Using this framework enabled us to factor out analyses that provide generally useful information into separate modules to facilitate the comprehension of the purity analysis. Figure 2 provides an overview of the analyses that are used by the purity analysis and their dependencies. We used independent analyses for determining a class' (im)mutability, the locality of fields, the freshness of return values, and to enable inference of escape information for local variables. In the following, when we talk about *the analysis*, we are always referring to the purity analysis *OPIUM*.

All created analyses use OPAL's three-address code representation of Java bytecode. The latter is based upon an intra-procedural data- and control-flow analysis and natively provides def-use information as well as refined local type information. Furthermore, lambda expression based invokedynamic calls are resolved using standard virtual methods calls and objects.

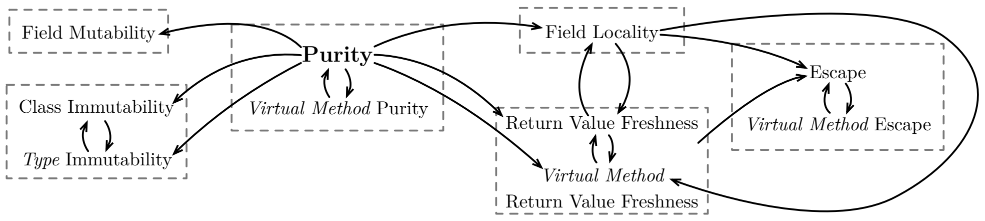

Figure 2: Dependencies between analyses used for purity analysis

4.1 Analysis Workflow

Our analysis (*Purity* in Figure 2) determines the purity level (cf. Figure 1) for each non-abstract method. It works as follows: When we start analyzing a method we assume that it is *pure* – the lattice's top value. We then check each statement of the method if it violates the currently assumed purity level and – if so – reduce the purity level to the next best state that is not violated. E.g., when the currently assumed level is *pure* and an exception object is created the level is decreased to *side-effect free*. Statements that access fields or other methods are initially ignored; instead a dependency on them is recorded. After analyzing all statements, the initial result, which consists of the currently assumed purity level and all dependencies, is passed to the framework. The information about the dependencies is used by the framework to call back the analysis when the required information regarding dependencies is updated. The analyses then updates its assumed purity level accordingly and also the set of (still) relevant dependencies and passes both back to the framework. The analysis of a method has completed if there are no more dependencies that may affect the assumed purity level; the later is then the final purity level. If a cycle is found by the framework, it is automatically resolved by assuming the current derived purity level for the methods within the cycle. The latter is correct as it is the best solution that takes all dependencies into account.

4.2 Effect of Instructions

In the following, we discuss the instructions that may affect a method's purity. Instructions related to mathematical operations and constants, type checks and casts, or to the control flow (e.g.,`add`, `if`, or `goto`) never affect a method's purity and thus are ignored.

Field Accesses: Field reads (`GetField` or `GetStatic`) introduce non-determinism when the read fields' values can change; i.e., when the fields are not (effectively) final. Therefore, the best possible purity level will be *side-effect free* unless the receiver object of the field access is local to the method. For example, accesses to a field of a newly created, non-escaping object are ignored. A field's mutability is determined by the respective analysis.

Note that it is sufficient to require that fields that are accessed are (effectively) final, this ensures that the read value is never changed. While the read value may be a reference to a mutable object or array, that mutability can only be observed if another field/array access is performed on the acquired reference.

Writing static fields (`PutStatic`) always reduces the method's purity to *impure*. Instance field writes (`PutField`) effect the purity if the written object is not local (cf. Locality). If the receiver of the field access is the self reference (`this`) of the method, the best purity level will be *external purity*. If the receiver is a formal parameter of the method, the method is at most *contextually pure*.

Array Accesses: We consider arrays as being objects where all fields (array entries) are mutable. `ArrayLoad` and `ArrayStore` instructions are hence handled equivalently to instance field accesses.

Synchronization: We treat explicit acquisitions and releases of monitors (`Monitorenter`, `Monitorexit`) as writes of an (implicit) field `monitor`.

Return from Method: A method which returns a reference is deterministic only if the returned object graph is guaranteed to be structurally equal (cf. Section 3) across method invocations. Otherwise, the best possible level is *side-effect free*. Structural equality is guaranteed if the returned reference is fresh and non-escaping or if the returned object is immutable. The latter property is derived by an independent class- and type immutability analysis.

Method Calls: The purity of the called method – independent of the underlying instruction (e.g. `StaticCall, VirtualCall`) – determines the purity of the caller as follows: If the callee's purity is *pure*, *side-effect free*, *domain-specific pure* or *impure*, the best possible purity level for the caller will be that of the callee. For an *externally pure* callee, the best possible level for the caller depends on the receiver object of the call. The caller can be *pure* if it is local and non-escaping. If it is the receiver object of the caller or one of its formal parameters, the caller's purity can not be better than *externally pure* or *contextually pure*, respectively. In all other cases, the caller must be *impure*. *Contextually pure* callee's are handled in the same way, except that all of their parameters, including their receiver object, must match the given condition. Callers with a combined purity level are handled like separate callees with the individual levels; e.g. a callee that is *domain-specific external side-effect free* requires that the caller is not better than each of *side-effect free*, *domain-specific pure* and the purity level that would be the result of an *externally pure* call on the same receiver object.

In case of a virtual method call, the callees are identified using a *Class Hierarchy Analysis*-based call graph. Results by Rountev [35] show that the use of a more complex call-graph algorithm has only minor impact on purity results.

We manually assigned purity levels to some native methods (e.g., `StrictMath.sqrt`, `System.arraycopy`) to improve the precision of the analysis. As other purity analyses [26, 39], we also use this mechanism to specify the following methods as *pure*: `hashCode`, `equals`, and `compareTo`. However, we do not handle `toString` specially (as others) since assigning correct purity levels for `StringBuilder` and `StringBuffer`'s append and `toString` methods suffices to correctly classify most `toString` methods.

Calls that are part of logging or console output are treated as *domain-specific*, *i.e., domain-specific pure* is used instead of the callee's actual purity level.

Allocation of Objects Allocations of objects in general do not influence a method's purity, besides the effect that invoking their constructor has according to the rules for method calls as above. The constructor of `Throwable` – the superclass of all exceptions – is, however, *impure* by definition because it invokes `fillInStackTrace` which might be impurely overridden by a subclass. This is a problem as exceptions occur frequently and should not in general result in *impurity*. We therefore treat `Throwable`'s constructor as *side-effect free*[2] and individually examine all subclasses of `Throwable` that override `fillInStackTrace`. Call-sites of exception constructors are treated as *domain-specific pure*.

4.3 Special Cases

Besides explicit statements, there are implicit effects on a methods purity that are also checked by the analysis.

Synchronized Methods: If a method is `synchronized` as such, the purity level will be at most *externally pure*; it is equivalent to writing an (implicit) field `monitor` on the method's receiver object.

Implicit Exceptions Exceptions may not only be allocated explicitly, but the Java VM also raises exceptions on several occasions, *e.g.,* a `NullPointerException` is raised when a method call's receiver is `null`. We recognize these implicit exceptions by examination of the control-flow graph (for exceptions that terminate the method's execution) as well as `CaughtException` statements. Those are again treated as *domain-specific pure*, as the constructors of all exception types that are potentially created by the Java Virtual Machine knowingly do not have any side-effects.

4.4 Locality

Modifications of objects that are constrained to the scope of a specific method can be ignored when computing the purity of such methods; in such cases the side-effect is confined and the method can still be pure. Such objects are called *Local* and in order to identify them, we implemented an escape analysis. The analysis derives three different escape values, namely *No Escape*, *Escape Via Return*, and *Global Escape* (cf. [10]). Objects with a lifetime that is bounded to the creating method's scope have the property *No Escape*. If such an object is returned by a method, the escape value is *Escape Via Return*. *Global Escape* is used for all other objects.

The escape analysis computes the respective property for each intra-procedural definition site. That is, for all formal method parameters, every object and array allocation site, all calls of methods that return an object and all field reads. The def-use information provided by OPAL is used to identify every use site and to propagate the effect of the use site back to the def-site. For example, if the only use-site is a return statement, the escape state of the respective object is *Escape Via Return*. The analysis is field-insensitive, i.e., whenever an object is stored in a field it assumes a *Global Escape*.

Using the results of the basic escape analysis, we identify local objects by examining all their definition sites. A reference is considered local if it is: (1) freshly allocated, (2) a *fresh return value* [22], or (3) a *local field* [33]. Whereas the locality for new allocations is trivially computed, we designed separate analyses – both depending on the escape analysis' results – that identify fresh return values and local fields. A method's return value is considered fresh if it is a newly allocated object or the result of a call to a method with a fresh return value. Furthermore, it must have the escape property *Escape Via Return*. When retrieving the return value freshness information for virtual calls, where the precise type of the receiver is not known, we aggregate the results for all potential call targets. A field is considered local [33] when its owning instance is local, all objects stored in the field are local, and no read value escapes (*No Escape*). The latter analysis requires special handling of `java.lang.Object`'s `clone` method which creates a shallow copy of an object – including all private fields of the object. Hence, a field might escape even if no `GetField` instruction is present. As mitigation, the analysis determines whether the object's class – whose field is under examination – overrides `clone` and stores a local object into the field of the new object. For classes that neither override `clone` nor implement the `Cloneable` interface, we can assume that the field is local if the class is final. In case of a non-final class, where the runtime type is precisely known, the field may also be considered to be local.

Furthermore, we have extended the field and return value analyses in order to deal with getter methods, i.e. instance methods that retrieve and return a field with `this` as receiver.

4.5 Threats to Soundness

Our analysis considers methods for which the implementation is not available, this in particular includes native methods, as *impure* unless explicitly specified differently. Therefore, it is sound even in the presence of local use of reflection or `sun.misc.Unsafe` which leads to the invocation of native methods. Non-local effects of such calls as well as reflective field write-accesses may however break the analysis' assumptions, and therefore result in unsound results.

5 EVALUATION

Our evaluation focuses on answering three research questions:

RQ1 Are the purity levels (cf. Figure 1) – in particular the new level *contextually pure* – found in real-world applications?

RQ2 Is an analysis that supports the identifications of all purity levels competitive w.r.t. precision and recall with the state-of-the-art which supports only specific level?

RQ3 Does the analysis scale to large applications?

For RQ1, we evaluated our analysis on the Oracle JDK 8 update 151, Scala 2.12.4, and the XCorpus [17]. The latter contains 76 programs: 70 programs from the Qualitas Corpus [40] and six additional ones that make use of modern dynamic language features of the Java VM. We had to exclude *jasperreports-1.1.0* due to invalid bytecode. For RQ2, we compared our analysis results against JPPA [38, 39], JPure [33], and ReIm [25, 26]. For this comparison we use the JOlden [9] benchmark. It consists of ten small Java programs from different domains that all tools were able to analyze. To answer RQ3, we report analysis and performance results for two real-world applications – *batik-1.7* and *xalan-2.7.1* from XCorpus – and compare them to ReIm. We chose these two applications,

[2]The method is not *pure* as the included stack trace depends on the currently executed method's context and not only on its parameters

Table 3: Purity results on XCorpus, JDK, and Scala

Application *Total methods*	**XCorpus** 469 727	**JDK** 253 282	**Scala** 174 881
Pure	73 701 (15.69%)	44 378 (17.52%)	28 502 (16.23%)
Domain-specific pure	9 628 (2.05%)	8 250 (3.26%)	6 625 (3.79%)
Side-effect free	45 268 (9.64%)	18 234 (7.20%)	10 793 (6.17%)
Domain-specific side-effect free	22 056 (4.70%)	14 717 (5.81%)	15 690 (8.97%)
Externally pure	19 467 (4.14%)	4 229 (1.67%)	2 519 (1.44%)
Externally side-effect free	2 380 (0.51%)	3 414 (1.35%)	82 (0.05%)
Domain-specific externally pure	639 (0.14%)	354 (0.14%)	11 (0.01%)
Domain-specific externally side-effect-free	2 467 (0.53%)	1 738 (0.69%)	2 339 (1.34%)
Contextually pure	4 (0.00%)	7 (0.00%)	0 (0.00%)
Contextually side-effect free	7 (0.00%)	48 (0.02%)	1 (0.00%)
Domain-specific contextually pure	522 (0.11%)	547 (0.22%)	31 (0.02%)
Domain-specific contextually side-effect free	1 523 (0.32%)	1 277 (0.50%)	80 (0.05%)
Impure	292 065 (62.18%)	156 089 (61.63%)	108 208 (61.87%)

because we were able to analyze them with ReIm. Neither JPPA nor JPure were able to analyze these applications.

5.1 RQ1 - Quantitative Results

The first column of Table 3 shows the different purity levels grouped by similar expressiveness. The last line lists the remaining impure methods. Columns two to four show (cf. Table 3) the total number of methods as well as the percentage of methods that were derived per purity level for XCorpus, JDK, and Scala.

The analysis shows that *pure* and *side-effect free* methods are commonplace. In all three cases ≈ 25% of all methods are in these two categories. Additionally, up to 12.76% of the methods are *domain-specific pure* or *side-effect free*. The analysis also identifies between 1.49% to 4.65% of all methods as being *externally pure/side-effect free* methods. In this case, we only found 82 *externally side-effect free methods* in Scala but a higher number of *domain-specific externally side-effect free* methods when compared to Java projects. This deviation is at least partly due to the different treatment of exceptions in these programming languages. In Scala exceptions never need to be caught and therefore less exceptions are explicitly caught. Hence, many more instructions may cause an abnormal return from a method. The same effect, albeit smaller, can be seen for *(domain-specific) side-effect free* methods in Scala.

Furthermore, we found multiple *contextually pure/side-effect free* methods. If we also take the *domain-specific* levels into account, we found 1879 methods in the JDK with this property, which is about 0.74% of all methods. Given that only 0.43% of all methods in the XCorpus and less than 0.1% in Scala have the respective level, the prevalence of these purity levels seems to be strongly dependent on the analyzed target. In general, the effects of *contextual purity* may improve when better contextual information is provided by supporting analyses. Based on the results, we conclude that all defined purity levels are actually found in real-world applications.

5.2 RQ2 - Comparative Analysis

We compared our analysis (*OPIUM*) against JPPA [38, 39], JPure [33] and ReIm [25, 26]. ReIm is the most recent one; representing the state-of-the-art. All three tools identify *side-effect free* methods and were downloaded from the authors' websites.

The sets of all methods that are analyzed by the tools have small differences. JPPA only analyzes methods transitively invoked by the main method, JPure and ReIm analyze all methods present in the source code and our approach analyzes all methods present in the class files which in particular includes static initializers and automatically generated default constructors. Furthermore, the reports of JPure and ReIm also include aggregated purity results for abstract methods. Aggregated information is in our case provided by the *Virtual Method Purity* analysis, not by the base purity analysis.

All produced analysis results are shown in Table 4. It lists for each of the JOlden projects the number of methods that each system identified as *side-effect free* (including *pure* methods for *OPIUM*) and the number of methods the system has analyzed. JPure failed to analyze TSP. For *OPIUM*, the table additionally gives the number of *pure* (including *domain-specific pure*) methods identified by our analysis and the number of methods with additional purity levels (i.e. *external* and *contextual purity* and variants thereof).

For this comparison, we treat the levels: *pure* and *side-effect free* as well as their *domain-specific* variants as *side-effect free*. Our analysis' results are competitive with ReIm and significantly outperforms JPPA and JPure. Hence, our analysis is competitive with state-of-the-art analyses for *side-effect free* methods. Additionally, our analysis identifies *pure* methods and a significant number of *externally* and *contextually pure/side-effect free* methods which are not found by the other projects including ReIm.

The differences between JPPA and JPure when compared to ReIm and our analysis in programs like *Power* and *TreeAdd* is due to a high number of constructors. These are not identified as *side-effect free* by JPPA's available implementation and JPure. For *TSP*, JPPA classifies several methods using `java.util.Random` as *side-effect*

Table 4: At least side-effect free (SEF) methods in JOlden

Program	BH	BiSort	Em3d	Health	MST	Perimeter	Power	TreeAdd	TSP	Voronoi
JPPA *#Analyzed methods*	59	13	20	26	31	36	29	5	14	60
At least Side-Effect Free methods	24	4	5	6	15	27	4	1	4	40
JPure *#Analyzed methods*	69	13	19	26	33	42	29	10	0	71
At least Side-Effect Free methods	10	3	1	2	12	31	2	1	–	30
ReIm *#Analyzed methods*	69	13	19	26	33	42	29	10	14	71
At least Side-Effect Free methods	33	5	8	11	16	38	10	6	1	47
OPIUM *#Analyzed methods*	70	15	23	29	36	45	32	12	16	73
At least SEF methods	33	7	8	13	19	38	11	7	4	49
Pure methods	11	6	5	9	8	21	7	6	3	12
Ext./Context. Pure/SEF methods	+18	+2	+3	+4	+6	+1	+13	+1	+0	+5

Table 5: At least side-effect free methods in Batik/Xalan

Program	Batik	Xalan
ReIm *#Analyzed methods*	16 029	10 386
At least Side-Effect Free Methods	6 072 (37.88%)	3 942 (37.95%)
Execution time	103s	140s
OPIUM *#Analyzed methods*	15 911	10 763
At least Side-Effect Free Methods	6 780 (42.61%)	4 390 (40.79%)
Pure methods	4 009 (25.20%)	2 492 (23.15%)
Ext./Context. Pure/SEF methods	+987 (6.20%)	+748 (6.95%)
Execution time	197s	187s

free, even though they are not – `Random` modifies global state when an instance is created – while ReIm fails to identify a pure constructor. We manually verified that all classifications performed by OPIUM were sound and identified further potential for improvements where we would be able to identify methods as pure if the supporting analyses would be more precise.

To foster an understanding of our analysis on real code, we compared it with ReIm on *Batik* and *Xalan*. The results are listed in Table 5. It again presents the number of methods identified as side-effect free (or even pure) by ReIm and our analysis alongside the number of analyzed methods. The number of *pure* methods and the number of additional, *externally* or *contextually pure/side-effect free* methods is given for *OPIUM*. The table also lists the execution time for both systems when analyzing *Batik* and *Xalan*.

On these applications, our analysis outperforms ReIm: We identify up to 5% more side-effect free methods and up to 7% of all methods as being externally or contextually pure/side-effect free methods. We can conclude that our analysis is competitive and also identifies all the defined purity levels

5.3 RQ3 - Performance Evaluation

The evaluation was performed on a Mac Pro with an Intel Xeon E5 processor (3 GHz) and 32 GB RAM running Lubuntu 17.10 with Oracle JDK 8 update 161; The Java VM was given 24 GB heap space.

The execution time for our and ReIm's purity analysis is listed in Table 5. It shows, that our (sequential) analysis, with 197s for Batik and 187s for Xalan, is slower than ReIm, which needs 103s and 140s to execute. The increase in runtime has multiple reasons: First, it also analyzes the JDK and all libraries required by the application whereas ReIm relies on a pre-annotated JDK. Second, the generation of three-address code – that eases the implementation and provides more precise type information – takes more than 45% of the overall execution time. However, our analysis derives more fine-grained results and identifies a significantly higher number of side-effect free methods than ReIm (cf. Table 5).

We can conclude, that the proposed analysis scales to large projects: The execution time is less than four minutes for ≈170 000 methods – including the JDK and library dependencies.

6 CONCLUSION

In this paper, we proposed a fine-grained unified lattice model for purity, covering all use cases from the literature. We also provide precise definitions for each lattice element to establish a common terminology. Furthermore, we implemented a scalable purity analysis which derives all properties of the lattice and which outperforms state-of-the-art purity analyses.

In future work, we will explore the use of the purity analysis as a supporting analysis for upstream ones. Additionally, we will leverage the modularity of the analyses to exchange the supporting analyses against more powerful ones to assess the effect of the latter when identifying pure methods.

OPIUM is available at `www.opal-project.de/Opium.html`.

ACKNOWLEDGMENTS

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. 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 CRISP.

REFERENCES

[1] Dotty documentation overview – Effect capabilities, 2018. Retrieved 2018-04-21.
[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.
[3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. *ACM Sigplan Notices*, 31(10):324–341, 1996.
[4] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *CSFW*, volume 2, page 253, 2002.
[5] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2004.
[6] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. Allowing state changes in specifications. *ETRICS*, 3995:321–336, 2006.
[7] W. C. Benton and C. N. Fischer. Mostly-functional behavior in Java programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 29–43. Springer, 2009.
[8] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 123–133. ACM, 2002.
[9] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 280–291. IEEE, 2001.
[10] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. *Acm Sigplan Notices*, 34(10):1–19, 1999.
[11] L. R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency and Computation: Practice and Experience*, 9(11):1031–1045, 1997.
[12] D. R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.
[13] V. Dallmeier. Static vs. dynamic purity analysis. *Dept. Comput. Sci., Saarland Univ., Saarbrucken, Germany, Tech. Rep. SU-CS-2013-012*, 2007.
[14] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24. ACM, 2006.
[15] Á. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422, pages 336–351. Springer, 2007.
[16] Á. Darvas and P. Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, 2006.
[17] J. Dietrich, H. Schole, L. Sui, and E. Tempero. XCorpus–an executable corpus of Java programs. *The Journal of Object Technology*, 16(4), 2017.
[18] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, 2003.
[19] M. Eichberg, F. Kübler, D. Helm, M. Reif, G. Salvaneschi, and M. Mezini. Lattice based modularization of static analyses. SOAP 2018, 2018.
[20] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 161–174. ACM, 2008.
[21] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. PLDI 2002: Extended static checking for Java. *ACM Sigplan Notices*, 48(4S):22–33, 2013.
[22] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction*, pages 82–93. Springer, 2000.
[23] S. Genaim and F. Spoto. Constancy analysis. *Formal Techniques for Java-like Programs (FTfJP)*, page 100, 2008.
[24] M. Hind and A. Pioli. Which pointer analysis should I use? In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 113–123. ACM, 2000.
[25] W. Huang and A. Milanova. ReImInfer: Method purity inference for Java. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 38. ACM, 2012.
[26] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *ACM SIGPLAN Notices*, volume 47, pages 879–896. ACM, 2012.
[27] R. Ierusalimschy and N. Rodriguez. Side-effect free functions in object-oriented languages. *Computer languages*, 21(3):129–146, 1995.
[28] A. Le, O. Lhoták, and L. J. Hendren. Using inter-procedural side-effect information in JIT optimizations. In *CC*, pages 287–304. Springer, 2005.
[29] G. T. Leavens and Y. Cheon. Design by contract with JML, 2006.
[30] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57. ACM, 1988.
[31] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical report, Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, 2003.
[32] D. A. Naumann. Observational purity and encapsulation. In *FASE*, volume 3442, pages 190–204. Springer, 2005.
[33] D. Pearce. JPure: a modular purity system for Java. In *Compiler construction*, pages 104–123. Springer, 2011.
[34] A. Potanin, J. Östlund, Y. Zibin, and M. D. Ernst. Immutability. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 233–269. Springer, 2013.
[35] A. Rountev. Precise identification of side-effect-free methods in Java. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 82–91. IEEE, 2004.
[36] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.
[37] A. Stewart, R. Cardell-Oliver, and R. Davies. Fine-grained classification of side-effect free methods in real-world Java code and applications to software security. In *Proceedings of the Australasian Computer Science Week Multiconference*, page 37. ACM, 2016.
[38] A. Sălcianu and M. Rinard. A combined pointer and purity analysis for Java programs. 2004.
[39] A. Sălcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, volume 5, pages 199–215. Springer, 2005.
[40] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas corpus: A curated collection of Java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336–345. IEEE, 2010.
[41] O. Tkachuk and M. B. Dwyer. *Adapting side effects analysis for modular program model checking*, volume 28. ACM, 2003.
[42] H. Xu, C. J. Pickett, and C. Verbrugge. Dynamic purity analysis for Java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 75–82. ACM, 2007.
[43] J. Zhao, I. Rogers, C. Kirkham, and I. Watson. Pure method analysis within Jikes RVM. *Proc. ICOOOLPS*, 2008.