

# A Software Product Line for Static Analyses

## The OPAL Framework

Michael Eichberg Ben Hermann

Technische Universität Darmstadt  
{lastname}@cs.tu-darmstadt.de

### Abstract

Implementations of static analyses are usually tailored toward a single goal to be efficient, hampering reusability and adaptability of the components of an analysis. To solve these issues, we propose to implement static analyses as highly-configurable software product lines (SPLs). Furthermore, we also discuss an implementation of an SPL for static analyses – called OPAL – that uses advanced language features offered by the Scala programming language to get an easily adaptable and (type-)safe software product line.

OPAL is a general purpose library for static analysis of Java Bytecode that is already successfully used. We present OPAL and show how a design based on software product line engineering benefits the implementation of static analyses with the framework.

**Categories and Subject Descriptors** F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis

**General Terms** Design, Languages, Program analysis

**Keywords** Static analysis, Design, Software Product Line Engineering, Abstract Interpretation

## 1. Introduction

When designing static analyses we aim for efficiency and scalability so that the analyses can tackle reasonable and therefore interesting problems. In order to achieve these design goals static analyses are usually tailored toward solving a single, specific set of problems and therefore often lack generality and reusability. In order to foster reusability and make specific analyses usable in a broader context, static analyses need to be more adaptable and require better support for variability without sacrificing performance.

A well-known approach to address variability in a managed fashion is *software product line engineering* (SPLE). We propose to design and implement static analysis frameworks as product lines in order to foster reuse of analysis components and allow for tailored but generally useful analyses.

In this paper we present OPAL, a framework for the static analysis of Java Bytecode which implements a software product line for the systematic creation of tailored static analyses. OPAL was designed to satisfy both fundamental requirements: (1) easy customizability and reusability as well as (2) performance and scalability. It uses state-of-the-art programming language abstraction from Scala to foster the development of new static analyses.

The OPAL Framework currently offers two variation points where analyses can be configured to specific requirements. First, the representation of Bytecode can be configured to the exact needs of the analysis in order to save resources and to support tools that have different requirements on the basic representation. Second, OPAL can be configured to run basic analyses in order to help higher-level static analyses by means of abstract interpretation.

The contributions of this paper are:

- An approach for designing static analysis frameworks based on software product line engineering.
- OPAL, a reference implementation for this design approach, which supports multiple representations for Java Bytecode as well as the configuration and adaptation of the performed static analyses to the needs of some user-developed higher-level static analysis.

The remainder of this paper is structured as follows. Motivating our work, we extend on related work in Section 2. In Section 3, we present a short introduction into software product line engineering. We provide a short introduction into the OPAL framework in Section 4. After that, we discuss OPAL's design w.r.t. its support for software product lines. The section ends with a discussion how it can be used to develop specifically tailored static analyses. In Section 6, we show an example where OPAL has already proven beneficial for the implementation of an analysis. We conclude the work in Section 7 with a summary and ideas for possible future work.

## 2. Related Work

In general, the idea of developing single, basic static analyses such that they are (re)usable is commonplace. But systematic reusability and composability of basic static analyses with well-/formally defined extension and variation points is not regularly addressed. An example of a step in that direction is, for example, the work on the generic framework for call graph algorithms done by Grove et al.[15]. They developed a framework that makes it possible to systematically configure the call graph construction algorithm. However, the primary purpose of that framework was to compare different call graph algorithms and not to provide a foundation for other developers of static analyses.

A second example of a framework that is related to our work is Julia developed by Fausto Spotto et al.[13]. This framework for

the abstract interpretation of Java Bytecode implements the core logic and enables the user to tailor the abstract interpretation to its needs. However, in that case the possible variability does not include changes to the base representation.

Soot [21] is a well-known static analysis framework for Java Bytecode. It uses various methods for configuration or extension and it is possible to configure Soot to run many different basic analysis. This can be done via a command-line or a programmatic interface. Additionally, the framework can be extended by adding custom code – so called transforms – to the execution packs Soot defines. Using this approach, Soot has been extended, e.g., to support complex data-flow analyses using the IFDS algorithm [7, 20] or to allow other Bytecode formats such as DEX [6]. Finally, multiple of those extensions can also be combined [12]. However, tailoring Soot’s existing analysis is generally done in an ad-hoc way and Soot uses multiple means to express the configuration of an analysis.

### 3. Software Product Line Engineering

Following [19], *software product line engineering* (SPLE) is a paradigm to develop software applications using platforms and mass customization. A platform in this sense is a set of components and interfaces that form a common structure to support managed variability.

A product of a product line is configured by a selection of features (variants) from a set of possible features (variation points). Individual features can be added to a product if they conform to the structure of managed variability the product line offers.

One of the most popular formal modeling techniques for software product lines are *Feature Models*. They organize features and their dependencies in a hierarchical fashion based on their granularity. The relationships between features can be expressed using the following primitives:

<b>Mandatory feature</b>	If a parent feature is present in the product, so must this feature.	
<b>Optional feature</b>	If a parent feature is present in the product, this feature may or may not be present.	
<b>OR relationship</b>	If a parent feature is present in the product, at least one of the child features has to be present.	
<b>XOR relationship</b>	If a parent feature is present in the product, exactly one of the child features has to be present.	

As in software product line engineering developers need to put effort into predictive rather than opportunistic software reuse. The advantages include significant improvements in cost and time-to-market. [22]

### 4. Overview of the OPAL Framework

OPAL[3] is a general purpose static analysis library for Java Bytecode that is developed with the goals of being easy to customize, extensible and scalable. OPAL supports Java 1.1 - 1.8 class files and facilitates the development of analyses ranging from prototyping ad-hoc, lightweight static analysis to the development of static analyses that need inter-procedural data- and control flow information.

The OPAL framework is the result of the experience gained while developing the Java Bytecode Framework BAT [10, 11] as

well as static analyses for Java Bytecode [17]. OPAL is already successfully used[14, 17]<sup>1</sup>. The overall design is also influenced by the design of the following frameworks for the static analysis/engineering of Java Bytecode: BCEL[2], ASM[1], Javassist[8] and Soot[4].

In contrast to the previous frameworks, OPAL is implemented in Scala and uses the language’s advanced features, to achieve the described design goals.

**Customizability and Extensibility** For these two design goals OPAL in particular relies on Scala’s support for mixin-composition by means of traits, the advanced type system (path-dependent types, self-type annotations and dependent method types) and Scala’s pattern matching.

**Scalability** This goal is achieved by two major design decisions:

First, by parallelizing all implemented analyses and by facilitating the development of static analyses that are parallelized. To this end, all core data-structures (e.g., the representation of the classes, the class hierarchy and the call graph) are thread safe. In case of OPAL, thread safety is primarily achieved by making the data-structures effectively immutable. A design that uses immutable data-structures is generally fostered by Scala, as Scala favors immutable data-structure over mutable ones. Hence, most of OPAL is thread-safe by design. This makes it possible to access the data-structures concurrently without any needs for further explicit synchronization.

Second, by facilitating the customization of OPAL, it is possible to adapt the framework to the *precise* needs of the analysis that will be developed. This generally helps to improve the overall memory- and runtime-performance and will be discussed next.

In the following, the features of the OPAL software-product line and how to customize OPAL to the needs of an analysis is shown. A detailed discussion of the performance and scalability is out of scope for this paper.

## 5. The OPAL Software Product Line

The Software Product Line implemented by OPAL basically consists of two major parts. The first one makes it possible to create very different kinds of representations for Java Bytecode and the second one makes it possible to customize and extend the performed basic static analyses to the specific needs of user-developed higher-level static analyses. Given that the second one analyzes a method’s implementation, it requires that the configuration of the first one contains the respective features. That means, it requires that the bytecode instructions of a method are represented using specific classes. Both parts are discussed in the next two sections.

### 5.1 Different Representations for Java Bytecode

As discussed in the introduction, the requirements of tools on the representation of Java Bytecode vary greatly. They range from representations using objects over XML to Prolog based representations. Support for this kind of variability is achieved by completely decoupling the infrastructure for reading in Java class files from the representation that is generated. Moreover, independent of the generated representation, the code for parsing class files can always directly be reused and adapted. This variability is achieved by following the approach proposed in [18]; the paper discusses – among others – family polymorphism in the context of Scala and how to achieve that by means of a component oriented programming model.

In OPAL this approach is reified by defining one *service component* for each major data structure of a Java class file (basically, the class file as such, the constant pool, a field, a method and each

<sup>1</sup> In the respective papers the framework was called BAT.

of the attributes of a class file). Each component (e.g., for processing a class file – cf. Listing 1) is responsible for reading in the respective data structure (Line 22) and to create the representation of the data structure using a factory method (Line 14). This approach enables us to completely abstract over the generated representation (Lines 2–7). The reading of other major data structures (Lines 24–28) is always delegated to the respective components (Lines 9–12). This is done using abstract methods which need to be implemented by other components.

```

1 trait ClassFileReader{
2   /* Abstract over the representation of the ... */
3   type ClassFile
4   type Constant.Pool
5   type Fields
6   type Methods
7   type Attributes
8   ...
9   /* Methods to read in the respective data structures. */
10  def Constant.Pool(in: DataInputStream): Constant.Pool
11  def Fields(in: DataInputStream, cp: Constant.Pool): Fields
12  def Methods(in: DataInputStream, cp: Constant.Pool): Methods
13  ...
14  /* Factory method to create a representation of a Class File. */
15  def ClassFile(
16    ... // Version information, defined type, etc.
17    fields: Fields,
18    methods: Methods,
19    attributes: Attributes)(
20    implicit cp: Constant.Pool): ClassFile
21
22  def ClassFile(in: DataInputStream): ClassFile = {
23    // read magic and version information
24    val cp = Constant.Pool(in)
25    // read access flags etc.
26    val fields = Fields(in,cp)
27    val methods = Methods(in,cp)
28    val attributes = Attributes(in,cp)
29    // call factory method
30    ClassFile(...,fields,methods,attributes)(cp)
31  }
32 }

```

**Listing 1.** Basic infrastructure for reading Java class files

To be able to read a Java class file it is necessary to plug all service components together (cf. Listing 3) such that all requirements of all components are satisfied. This requires that all abstract types are made concrete and that the factory methods are correspondingly implemented. This is generally done in traits that extend the *XYZReader* traits and are called *XYZBinding*. For example, Listing 2 shows the trait with the final type binding (Line 4) and factory method for creating representations of *Methods* (Line 5-9).

```

1 trait MethodsBinding extends MethodsReader {
2   this: ConstantPoolBinding with AttributeBinding =>
3
4   type Method_Info = de.tud.cs.st.bat.resolved.Method
5   def Method_Info(
6     accessFlags: Int, name: Int, descriptor: Int,
7     attributes: Attributes)(
8     implicit cp: Constant.Pool): Method_Info =
9     create Method representation
10 }

```

**Listing 2.** Final type binding for creating Method representations

The final composition is then supported by Scala by means of mixin-composition and ensures that only compatible *service components* are plugged together and that no requirements remain unsatisfied. A minimal, valid configuration is shown in Listing 3. It can be used to read a class file’s public interface if the analysis

does not need information about the implementation of the respective classes. This is often the case for architecture analyses. In that case detailed information about the implementation of the libraries used by the application/library is not required. A configuration that completely reifies all standard information is shown in Listing 4. An excerpt of the feature diagram related to the variability of the first part is outlined in Figure 1. The diagram shows the variability discussed in the code. Additionally, it shows that we actually have two implementations of a *MethodsReader*; one that creates a representation that completely abstracts from the constant pool (starting with “*resolved.MethodsBinding*”) and which is comparable to the representation used by, e.g., BCEL and one that is a raw representation that keeps the references to the constant pool (called “*native.MethodsBinding*”).

```

1 class Java7ClassFilesPublicInterface
2   extends ConstantPoolBinding
3   with InterfacesBinding
4   with FieldsBinding
5   with MethodsBinding
6   with AttributesReader
7   with SkipUnknown.attributeReader
8   with AnnotationsBinding
9   with InnerClasses.attributeBinding
10  with ClassFileBinding
11  // further attributes related to a class' public interface

```

**Listing 3.** A complete configuration to read in the public interface of .class files

```

1 class Java7ClassFiles extends Java7ClassFilesPublicInterface
2   with CodeAttributeBinding
3   with StackMapTable.attributeBinding
4   with LineNumberTable.attributeBinding
5   with LocalVariableTable.attributeBinding
6   with BootstrapMethods.attributeBinding
7   // further “code” related attributes

```

**Listing 4.** Completely reifying a .class File

Given the proposed approach, it is directly possible to specify which information should be made available in which way. Additionally, it is possible to selectively adapt a generated representation by exchanging the component for reading a method’s *Code Attribute* (the body of a method) and to reuse all other components. This could be useful if we want to directly transform the stack-based bytecode representation in a higher-level representation such as single static assignment form[9]. Finally, it is possible to reuse the complete infrastructure for parsing class files to create completely different representations (e.g. XML or Prolog based).

## 5.2 Extensible and Configurable Static Analysis

The second part of the software product line enables the user to adapt the basic static analyses performed by the framework to the needs of some higher-level static analyses.

To this end, OPAL implements a highly-configurable framework for the (abstract) interpretation of Java Bytecode that facilitates the development of lightweight static analyses for bug detection or for verifying certain properties. As in the previous case, each major building block is represented as a component. In this case, however, a component is responsible for performing all relevant computations related to a specific set of bytecode instructions. Furthermore, Scala’s *virtual classes pattern*<sup>2</sup> is used in its full entirety. This pattern enables a form of family polymorphism and has

<sup>2</sup><https://wiki.scala-lang.org/display/SIW/VirtualClasses>

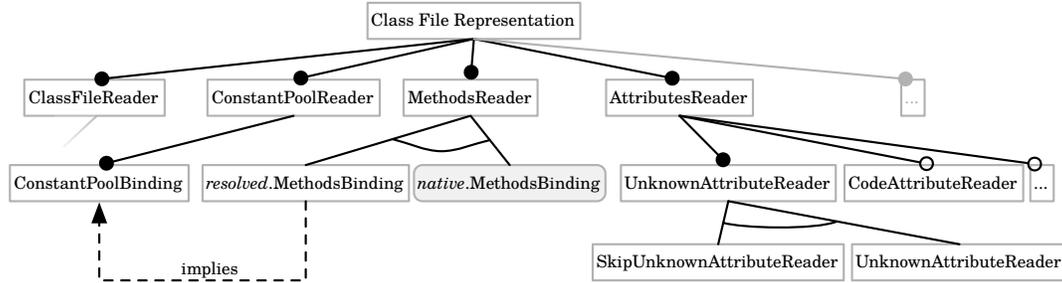


Figure 1. Excerpt of the Feature Diagram related to supporting different representations

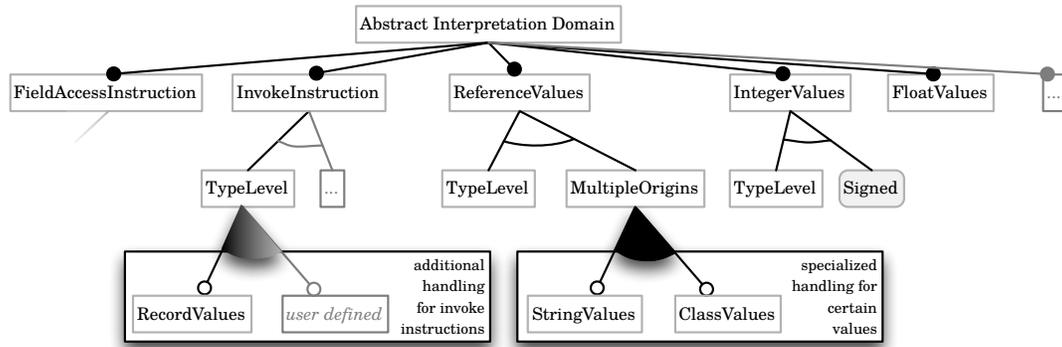


Figure 2. Excerpt of the Feature Diagram related to the configuration of an abstract domain

similar advantages as offered by *native* virtual classes as available, e.g., in CaesarJ<sup>3</sup>[5].

For example, one component (called a *Domain* in the framework) – shown in Listing 5 – is responsible for performing all computations related to Long values and each component basically contains two major parts. The first part (Lines 3-8) refines the representation for handling Long values. In the shown case, only the type information is encoded, but it is possible to encode “arbitrary” information, such as the value’s sign or the value’s current range. The second part (Lines 10-19) implements all bytecode instructions related to the respective value and the chosen representation. The conversion instructions (Lines 17-19) are typically generically implemented using the generic factory methods declared by Domain. Finally, each component/domain is responsible for implementing its related factory methods (Lines 21-24). Hence, in this design, the trait Domain has the role of the central coordinator that declares *all* methods needed by the framework for the interpretation of the bytecode and each component implements a cohesive subset of the methods as just discussed. Additionally, each domain can provide additional functionality that is needed by subsequent analyses, but which is not required by the framework.

```

1 trait DefaultTypeLevelLongValues extends Domain {
2   type DV = DomainValue
3   case object ALongValue extends Value with IsLongValue {
4     this: DV =>
5     override def computationalType = ComputationalTypeLong

```

<sup>3</sup>Compared to languages with native support for virtual classes, Scala’s *virtual classes pattern* is syntactically far more verbose and requires the user, e.g., to perform the deep mixin-composition of the virtual classes manually. This limits its scalability. Nevertheless, the same level of type safety can be achieved and overall the pattern has proven to be suitable for our purposes.

```

6   override def doJoin(pc: PC, value: DV): Update[DV] =
7     NoUpdate
8   }
9
10  override def lcmp(pc: PC, v1: DV, v2: DV): DV = ...
11  override def lneg(pc: PC, v: DV) = ...
12
13  override def ladd(pc: PC, v1: DV, v2: DV): DV = ...
14  ... other mathematical instructions related to long values
15  override def lxor(pc: PC, v1: DV, v2: DV): DV = ...
16
17  override def l2d(pc: PC, v: DV): DV = ...
18  override def l2f(pc: PC, v: DV): DV = ...
19  override def l2i(pc: PC, v: DV): DV = ...
20
21  override def LongValue(pc: PC): DV =
22    ALongValue
23  override def LongValue(pc: PC, value: Long): DV =
24    ALongValue
25 }

```

Listing 5. Component/Domain related to computations with Long values

The variation points that results from the proposed componentization<sup>4</sup> are shown in the feature diagram Figure 2. We have one variation point per Java Virtual Machine (JVM) level type [16] (*reference values*, int, long, float, double) as well as further variation points for further cohesive sets of instructions (e.g., method invocations, field access, synchronization, ...). For each variation point one to more implementations exist that enable a fine grained control over the overall precision of the analysis. In all cases a basic implementation exists that deals with the respective instructions at

<sup>4</sup>Other partitionings are also supported by the framework, but are out of scope for this paper.

the JVM type level. But, as shown in the feature diagram, more advanced features are also readily available. For example, in case of reference values it is possible to precisely track `java.lang.String` and `java.lang.Class` values, which is often useful for analyses that need to consider Java Reflection.

As previously, to get a concrete Domain that can be used by OPAL to interpret some Java bytecode, it is sufficient to do a mix-in-composition of the desired domains (cf. Listing 6).

In general, a complete domain as shown in Listing 6 can be used in two different ways: First, to directly perform an abstract interpretation of a method and to analyze the results afterwards. Specifically this means, as soon as the abstract interpretation has completed, it is possible to analyze the abstract values (operands) of each instruction and to detect, e.g., that a certain if-condition always or never holds. Second, to further adapt the domain to perform some computations on the fly while the interpretation is going on.

```

1 class BaseDomain extends Domain
2   with DefaultDomainValueBinding
3   with Configuration
4   with DefaultTypeLevelReferenceValues
5   with DefaultTypeLevelIntegerValues
6   with DefaultTypeLevelLongValues
7   with DefaultTypeLevelFloatValues
8   with DefaultTypeLevelDoubleValues
9   with DefaultIntegerValuesComparison
10  with TypeLevelFieldAccessInstructions
11  with TypeLevelInvokeInstructions
12  with ProjectBasedClassHierarchy
13  with IgnoreMethodResults
14  with IgnoreSynchronization {
15  type Id = String
16  def id : Id = "TypeLevelDomain"
17 }

```

**Listing 6.** Final configuration of a Domain

```

1 class CHACallGraphDomain(
2   val project: Project[URL],
3   val theClassFile: ClassFile, val theMethod: Method)
4   extends BaseDomain {
5   type Id = Method
6   def id = theMethod
7
8   abstract override def invokevirtual(
9     pc: PC, declaringClass: ReferenceType,
10    name: String, descriptor: MethodDescriptor,
11    operands: List[DomainValue]): MethodCallResult = {
12    val result = super.invokevirtual(
13      pc, declaringClass, name, descriptor, operands)
14    ... construct call graph ...
15    result
16  }
17  ... handling for other invoke instructions ...
18 }

```

**Listing 7.** A Domain for calculating a Call Graph

A corresponding example is shown in Listing 7; the call graph is iteratively computed whenever an `invoke(virtual, special, interface)` (Lines 8-16) instruction is encountered. In that case, the general handling of `invoke` instructions is implemented by `BaseDomain` and the `CHACallGraphDomain` just intercepts (Line 8) the respective calls, but relies on the `BaseDomain` (Line 12) for computing the result.

## 6. Discussion

To validate the general approach, we have implemented an analysis two times, once using SOOT and the other using OPAL. The analy-

sis searches for specific instances of a confused-deputy. In this case we search for calls to `Class.forName(...)` that are executed by *privileged* code of the JDK, but where the class name is determined by an unprivileged caller of the privileged code. This enables unprivileged callers to load potentially harmful classes. The latter then have the same rights as privileged classes. This effectively circumvents the Java sandbox. The analysis is motivated by a vulnerability of the Java platform that is listed in the Common Vulnerabilities and Exposures Directory under the identifier CVE-2013-4681.

The SOOT[21] based version of the analysis uses an implementation of the IFDS algorithm [20] as its foundation. In case of OPAL, we use the abstract interpretation framework as a foundation which is tailored to the specific needs of the analysis and exploits OPAL’s SPL features.

```

1 trait TaintAnalysisDomain[Source]
2   extends Domain
3   with DefaultDomainValueBinding
4   with Configuration
5   with DefaultStringValueBinding // handles all reference values
6   with DefaultTypeLevelIntegerValues
7   with DefaultTypeLevelLongValues
8   with DefaultTypeLevelFloatValues
9   with DefaultTypeLevelDoubleValues
10  with TypeLevelFieldAccessInstructions
11  with TypeLevelInvokeInstructions
12  with ProjectBasedClassHierarchy[Source]
13  with IgnoreMethodResults
14  with IgnoreSynchronization
15  with Report {
16  type Id = CallStackEntry
17  // Further Customization
18  // ( ≈635 lines of code)
19 }

```

**Listing 8.** Base configuration of a Domain used for taint analyses.

In Listing 8 the *Domain* that is used to implement the described analysis is shown. We use – when compared to the domain used for constructing the call graph (Listing 7) – a different domain for reference values. This analysis needs more precise knowledge about `StringValues` and for that reason uses a domain that can more precisely track respective values (Line 5 in Listing 8).

Both analyses do identify the same callers and have roughly comparable runtimes<sup>5</sup>. In case of OPAL the analysis is implemented in ≈650 lines of Scala code. The Java-based implementation in SOOT required ≈5000 lines of code. Given these numbers, we are convinced that the approach implemented by OPAL provides a good foundation for the efficient development of static analyses.

All line number metrics provided in this section are line numbers excluding comments.

## 7. Conclusion

The OPAL framework, as presented, is a significant step in the direction of a software product line for static analyses that offers users *managed variability* and *systematic adaptability* of the framework to their needs. The framework already offers a large number of variation points and enables the user to select from various ready-to-use features per variation point. This makes it possible to use the framework for analyses ranging from simple metrics over architecture validation to selected control- and data-flow analyses while only paying for features that are actually required. Additionally, the framework is generally open for extension and enables the user to implement additional features on its own.

<sup>5</sup> The times are not directly comparable as SOOT does various additional analyses, e.g., transformation into SSA form, that are not done by OPAL.

We are convinced, that the presented approach delivers an unprecedented level of *systematic* adaptability that scales to further use cases. However, further extensions of the software product line will be done w.r.t. the support for more complex inter-procedural analyses (e.g., points-to analyses). Addressing these issues is part of ongoing and future work.

## Acknowledgments

This work was supported by the German Ministry of Research and Education (BMBF) within EC SPRIDE.

## References

- [1] Asm, 2014. URL <http://asm.ow2.org>.
- [2] Byte code engineering library (BCEL), 2014. URL <http://commons.apache.org/proper/commons-bcel/>.
- [3] Opal, 2014. URL <http://www.opal-project.de>.
- [4] Soot: a Java Optimization Framework, 2014. URL <http://www.sable.mcgill.ca/soot/>.
- [5] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of caesarj. In A. Rashid and M. Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-32972-5. . URL [http://dx.doi.org/10.1007/11687061\\_5](http://dx.doi.org/10.1007/11687061_5).
- [6] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.
- [7] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *1st ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis*, pages 3–8, 2012. . URL <http://www.bodden.de/pubs/bodden12inter-procedural.pdf>.
- [8] S. Chiba. Javassist, 2014. URL <http://www.javassist.org/>.
- [9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991. ISSN 0164-0925. . URL <http://doi.acm.org/10.1145/115372.115320>.
- [10] M. Eichberg, M. Kahl, D. Saha, M. Mezini, and K. Ostermann. Automatic incrementalization of prolog based static analyses. In M. Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 109–123. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-69608-7. . URL [http://dx.doi.org/10.1007/978-3-540-69611-7\\_7](http://dx.doi.org/10.1007/978-3-540-69611-7_7).
- [11] M. Eichberg, M. Monperrus, S. Kloppenburg, and M. Mezini. Model-driven engineering of machine executable code. In T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, editors, *Modelling Foundations and Applications*, volume 6138 of *Lecture Notes in Computer Science*, pages 104–115. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-13594-1. . URL [http://dx.doi.org/10.1007/978-3-642-13595-8\\_10](http://dx.doi.org/10.1007/978-3-642-13595-8_10).
- [12] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, 2013. URL <http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf>.
- [13] S. Genaim and F. Spoto. Information flow analysis for java bytecode. In *VMCAI*, pages 346–362, 2005.
- [14] P. G. Giarrusso, K. Ostermann, M. Eichberg, R. Mitschke, T. Rendel, and C. Kästner. Reify your collection queries for modularity and speed! In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development, AOSD '13*, pages 1–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1766-5. . URL <http://doi.acm.org/10.1145/2451436.2451438>.
- [15] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, Nov. 2001. ISSN 0164-0925. . URL <http://doi.acm.org/10.1145/506315.506316>.
- [16] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java™ Virtual Machine Specification*. Oracle America, Inc., java se 7 edition edition, July 2011.
- [17] R. Mitschke, M. Eichberg, M. Mezini, A. Garcia, and I. Macia. Modular specification and checking of structural dependencies. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development, AOSD '13*, pages 85–96, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1766-5. . URL <http://doi.acm.org/10.1145/2451436.2451448>.
- [18] M. Odersky and M. Zenger. Scalable component abstractions. *SIGPLAN Not.*, 40(10):41–57, Oct. 2005. ISSN 0362-1340. . URL <http://doi.acm.org/10.1145/1103845.1094815>.
- [19] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering*, volume 10. Springer, 2005.
- [20] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.
- [21] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.
- [22] F. J. van der Linden, K. Schmid, and E. Rommes. *Software product lines in action*. Springer, 2007.