Self-adaptive static analysis

Eric Bodden
Paderborn University & Fraunhofer IEM
eric.bodden@upb.de

ABSTRACT

Static code analysis is a powerful approach to detect quality deficiencies such as performance bottlenecks, safety violations or security vulnerabilities already during a software system's implementation. Yet, as current software systems continue to grow, current static-analysis systems more frequently face the problem of insufficient scalability. We argue that this is mainly due to the fact that current static analyses are implemented fully manually, often in general-purpose programming languages such as Java or C, or in declarative languages such as Datalog. This design choice predefines the way in which the static analysis evaluates, and limits the optimizations and extensions static-analysis designers can apply.

To boost scalability to a new level, we propose to fuse static-analysis with just-in-time-optimization technology, introducing for the first time static analyses that are managed and inherently self-adaptive. Those analyses automatically adapt themselves to yield a performance/precision tradeoff that is optimal with respect to the analyzed software system and to the analysis itself.

Self-adaptivity is enabled by the novel idea of designing a dedicated intermediate representation, not for the analyzed program but for the analysis itself. This representation allows for an automatic optimization and adaptation of the analysis code, both ahead-of-time (through static analysis of the static analysis) as well as just-in-time during the analysis' execution, similar to just-in-time compilers.

ACM Reference format:
Eric Bodden. 2018. Self-adaptive static analysis. In *Proceedings of 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18 Companion),* 4 pages.
https://doi.org/10.1145/3183440.3183462

1 INTRODUCTION

Over the past decades, static code analysis has made significant progress, and has seen many novel applications: originally used mainly for the purpose of ahead-of-time program optimization [18, 19], it has now become also a common tool for program understanding as well as for finding software quality defects, in particular security vulnerabilities [2, 10, 22]. This success is due to decades of static-analysis research, which yielded the discovery of novel algorithms, data structures and design principles that make static analyses more precise and scalable than ever before. [9, 15–17, 21]

Yet at the same time, the size of software systems has grown immensely. Hence, while the progress in static-analysis research is significant in absolute terms, one must fear that nevertheless the technology will always lack behind the software applications' increase in size and complexity. To break this barrier, one requires nothing short of a breakthrough in static-analysis technology.

Such a break-through is currently hindered by the fact that, so far, all known static-analysis tools have been implemented by hand, and in general-purpose programming languages such as Java or C/C++, or in some cases partly in Datalog. Most static analyses require only a limited expressiveness, and thus often can be expressed as pushdown-problems [13] or even graph-reachability problems [12], certainly most often do not require a Turing-complete language. Since optimizations for pushdown automata and graph algorithms are well studied, one would think it possible to just apply powerful automated optimizations to such analyses. Yet, the current state of the art is to implement static analyses themselves in general-purpose programming languages. In our view, those languages are too expressive: they are, in fact, Turing complete, which greatly hinders powerful automated optimizations of the static analyses.

In this work we propose a novel fusion of static-analysis and just-in-time-optimization technology, yielding static analyses that are inherently self-adaptive, and use this self-adaptivity for self-optimization. Current analyses are not self-adaptive because their evaluation strategy is, at least for the most part, hard-coded. While analysis implementations might select among a set of multiple pre-defined evaluation strategies depending on the analysis problem and analyzed application at hand, the possible choices and the selection strategies themselves are fixed. Moreover, once a strategy has been selected, it is executed by instantiating pre-defined static-analysis components.

We instead envision a solution that produces for each concrete analysis problem a highly customized and optimized static-analysis implementation, specifically tailored to the problem at hand. Moreover, the solution should have the ability to re-adapt and thus further optimize this implementation based on an introspection into the analysis' own execution. This is what we mean when we speak of self-adaptivity, and this is where lessons learned from research on just-in-time optimization will be useful. Enabling such self-adaptivity requires one to design and implement the analysis according to a completely novel engineering methodology, a description of which is the core contribution of this paper.

Developing a working system fully implementing the idea we propose is a multi-person-year effort. In this paper we restrict ourselves to explaining the core idea and to posing the main research challenges one needs to address to obtain a working solution. That way we hope that the software engineering research community will join us in our quest for an optimal solution strategy.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3183462

Section 2 presents the core concepts and challenges of our proposal, while Section 3 situates the proposal into related work. Section 4 concludes.

2 CORE CONCEPTS AND CHALLENGES

The main objective of this work is to enable self-adaptive, self-optimizing static analyses. This, in turn, requires one to design and implement the analysis according to a completely novel engineering methodology, yielding an architecture outlined in Figure 1. The architecture fuses a number of concepts known from the area of static program analysis with those of just-in-time program optimizations, as we know it from application-level virtual machines. Importantly, though, the concepts have been fused such that not the program under analysis is the one that's being optimized but instead *the just-in-time-optimization targets the static analysis itself!*

In the following we outline the various core concepts and objectives that Figure 1 presents.

① *Declarative definition language for static analysis.* At the core of our proposal is the following paradigm shift: to represent the program analysis itself not in a general-purpose programming language such as Java or C/C++, (nor a general-purpose logic programming language such as Datalog) but rather in a domain-specific language that is optimally amenable to domain-specific optimizations, i.e., optimizations that are only correct when taking into account specific knowledge about the domain of static program analyses. To give an example, some data-flow functions are distributive with respect over the merge operator, in which case one can solve them more efficiently if this fact is recognized. [12] The challenge in designing such a language is that it must have exactly the right level of expressiveness: it must be expressive enough to cover a reasonably broad set of possible static-analysis clients, yet at the same time its expressiveness must be restricted enough such as to allow for powerful automated domain-specific optimizations. Moreover, the language must be easy enough for static-analysis users, i.e., typically for software developers, to write and understand, yet at the same time must be machine-readable as well. Using the language, it should be possible to conveniently express also large sets of static-analysis rules, as is common, for instance in security code analysis tools, which commonly check programs against hundreds of vulnerability patterns.

Challenge 1: The design and implementation of a domain-specific language (DSL) for expressing static-analysis rules.

② *High-level intermediate representation.* While the DSL above must be easy for humans to understand, to enable automated optimizations of the static analysis we instead envision a dedicated domain-specific intermediate representation (IR) of the static analysis, which is not designed to be optimally understandable by humans, but instead to be optimally amenable to domain-specific optimizations, i.e., optimizations that are only correct when taking into account specific knowledge about the domain of static program analyses. The focus of the proposed high-level intermediate representation is on optimizing static-analyses *ahead of time*, i.e., prior to their execution. Similarly to query optimizations in database research, such an IR will allow one to exploit synergies between similar analysis rules, and to find an optimal evaluation strategy—at this point independent of the analyzed program.

Challenge 2: The design, and implementation of a high-level intermediate analysis representation, with the goal to allow ahead-of-time analysis optimizations.

③ *Low-level intermediate representation.* The above ahead-of-time optimizations have the goal to improve analysis performance (while maintaining precision) *without* taking the analyzed program into account. Yet, previous experience shows that the analyzed program can have a great influence on what is the optimal configurations for a static analysis of that program. [3, 15] We hence desire a mechanism to allow domain-specific optimizations that take program characteristics into account, allowing the analysis configuration to be optimally tuned to the analyzed program while the analysis is being conducted. This requires one to alter the analysis execution just-in-time, or at least from one execution to the next. While in theory this might be able by altering its high-level intermediate representation (Challenge 2), it is likely that one can benefit from an additional low-level representation that is less declarative but closer to the analysis' actual execution. This also gives one the opportunity to combine in the same representation not just aspects of the analysis but also aspects of the program to be analyzed. Opposed to the high-level IR, to allow for just-in-time adaptation, for the low-level IR it must be possible to alter the analysis representation in place, potentially re-generating analysis code in the fly, similarly to how current virtual machines re-generate program code at runtime.

Challenge 3: The design, and implementation of a low-level intermediate representation for just-in-time optimizations during analysis execution, combining both aspects of the analysis and the program to be analyzed.

④ *Static-analysis profiler.* A self-optimizing static analyzer, but also a human static-analysis expert, will require deep insights into the analysis' own execution, and in particular its performance hotspots. The fact that the analysis is self-adaptive actually makes this harder than normal, as the analysis code that actually executes is not code hand-written by the analysis designer, but instead code generated from the analysis' intermediate representation. Such a self-adaptive design thus threatens to lose the link between the analysis definition and its execution. To address this challenge, we identify the objective of designing, implementing and evaluating a dedicated profiling tool for self-adaptive static analyses. The profiling tool is meant to automatically identify execution hotspots that cause the analysis performance to degrade, for instance regions of the analyzed program that cause the analysis to iterate for exceedingly long times, or cause it to consume unusually large amounts or memory. The profiler should moreover link back those performance hotspots to the relevant fragments of the static analysis responsible for those parts of the analysis execution, in the different IRs as well as in the original analysis definition in our novel DSL. The profiler also directly links back to our JIT analysis engine, which we describe next.

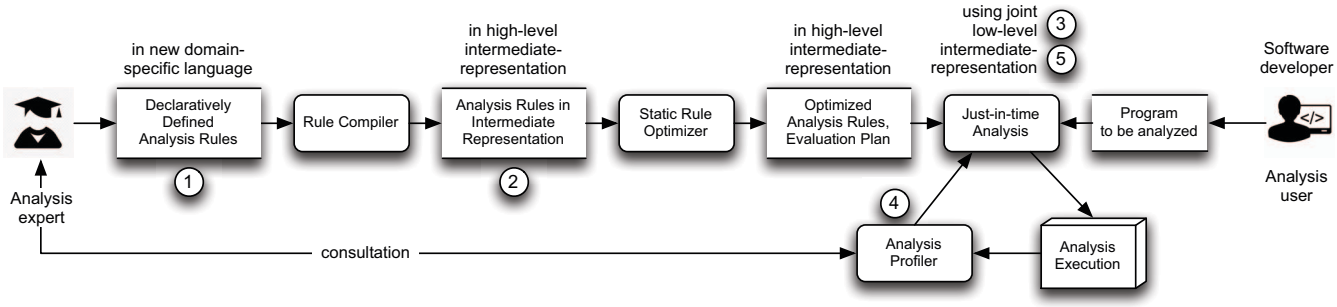

Figure 1: Workflow of the envisioned self-adaptive static analysis

Challenge 4: The design and implementation of a profiling tool for static-analysis runs, providing traceability into the different analysis representations.

⑤ *Automated just-in-time optimization.* The final step to towards our main objective is to develop an optimization engine that uses domain knowledge about the static analysis it executes, paired with information from the profiled analysis run, to determine more optimal analysis execution strategies, and to trigger those strategies through an automated self-adaptation. We envision this engine to effectively implement a control loop, in which profiling information is continuously processed to determine the optimality of the current strategy, and to determine other strategies that might be more efficient (or which may save more memory) during the remainder of the execution and/or during the next analysis run. One challenge in this space is to find the right machine learning algorithms to create a reasonable situational awareness of the executing analysis, and to draw the right conclusions from the observed profiles. Another challenge lies in the analysis adaptations. Some simple adaptations are implemented easily. For instance one can change a taint analysis from forward to backward if encountering in a program significantly more sources than sinks, thus yielding an analysis runs with a comparatively low number of taints. Or else, when computing a constant propagation, if observing that parts of a program conduct linear arithmetic only, one can use an efficient IFDS-based [12] tabulation solver to solve a linear-constant propagation problem for those program parts. Some adaptations, however, might involve more low-level adaptations of the executing analysis. For instance, one might think of actually implementing parts of the analysis using certain data structures, either tuned for runtime or memory efficiency, depending on where more efficiency is required. To this end, the optimization engine must be able to closely interact with the low-level intermediate representation (Challenge 3).

Challenge 5: An optimization engine that gradually optimizes a given analysis problem for precision and performance, based on profiled analysis runs.

3 STATE OF THE ART AND RELATED WORK

We next describe how related work from the state-of-the-art literature can fuel the research we propose here. Space restrictions force us to only discuss the most related areas of research.

Application-level virtual machines. In terms of its architecture, the solution we propose here has a strong similarity with the architecture of application-level virtual machines such as the Java virtual machine (JVM). Their just-in-time compilers (JIT) profile the hosted applications' execution to optimize that very same execution on the fly. Some virtual machines further include ahead-of-time optimizations and pool optimized code for efficient reuse on later runs.[5] And also here one can observe a tradeoff between automation and understandability: debugging and optimizing virtual machines is notoriously hard, which is why researchers have developed dedicated tools for this purpose.[20] Those tools typically visualize the executing program's code in the different intermediate representations on which the JVM itself operates. The main difference between a JVM and its JIT to our proposal is that here we are not executing general-purpose Java code but instead a specific static analysis. Moreover, we seek to have domain-specific representations of this static analysis at all levels of its execution. Thus, while one might, in fact, reuse ideas regarding the architecture and design of application-level virtual machines, the specific intermediate representations, the transformations between them, and the optimizations within them will greatly differ from those of existing approaches.

Declarative languages used for static analysis. Not all static analysis are implemented in Turing-complete general-purpose languages. The static-analysis framework Doop [4], for instance, implements its static analyses in the logic programming language Datalog. Using Datalog gives users the great advantage that the implemented static-analysis rules are relatively simple to write, read and reason about. The declarative nature of the language also means that—in theory—users need to not be concerned with how the rules are evaluated, as this decision is entirely left to the Datalog engine. Yet, as past experience with Doop and similar approaches has shown, to make Datalog-based analyses truly efficient, one still requires optimizations on two levels.

First, automated optimizations on the level of the Datalog language itself. Doop is frequently used in combination with highly optimized Datalog solvers such as LogicBlox [1] or Souffle[6]. The latter is a Datalog solver specifically designed for the purpose of supporting static analyses: it translates the datalog rules into C code implementing a highly optimized Datalog solver for the particular rule set at hand. To some extent Souffle is thus similar to what we

propose here, but the main difference of Souffle and all other Datalog engines, compared to what we propose here, is that Datalog, albeit being a logic programming language, is still a general-purpose language. It is not domain-specific, and thus has no domain-specific constructs that would make the static-analysis solution particularly efficient to compute. In result, while users benefit from the declarative nature of the language, they only benefit from automated general-purpose Datalog optimizations, which are limited because they lack domain knowledge. A particular limitation of Datalog, namely the lack of being able to express fixed-point iterations, which are commonplace in static analysis, triggered the incarnation of the novel logic programming language Flix [11]. Flix is essentially an extension of Datalog with certain primitives that allows analysis writers to directly express fixed-point computation. Yet, the tool that currently implements Flix is, again, nothing more but a general-purpose solver for the Flix language. It is not a static-analysis tool, and does not support any kind of further domain-specific optimizations.

Conceptual analysis frameworks. Some domain-specific optimizations that a just-in-time optimizer should consider for static analysis can be obtained by choosing the optimal *analysis framework*. By analysis framework we here mean a framework in the conceptual sense. For instance, two major conceptual frameworks are known to compute correct solutions for context-sensitive inter-procedural analysis problems: the so-called *call-strings approach* and the *functional approach* [14]. The call-strings approach has the advantage that it can be applied to pretty much any static data-flow analysis, specifically any such analysis that fits the monotone framework [7]. Yet, it has the drawback that it could analyze callee procedures while distinguishing more contexts than necessary, thus unnecessarily wasting computation time. Moreover, one must bound the amount of context the analysis uses, and choosing the bound poorly might jeopardize performance, precision and even correctness [8]. The functional approach has the advantage of providing unlimited context-sensitivity (hence no bound is required), yet is only tractable for analysis problems whose merge operator is set union and whose flow functions distribute over this merge operator—so-called IFDS or IDE problems[12]. In cases where the analysis problem fits such a framework, one has the advantage that one can compute precise solutions (equivalent to the theoretically optimal but generally uncomputable MOP solution [8]), and moreover that highly efficient solution algorithms exist. Moreover, in recent work, we were able to show that it is sometimes possible to decompose such analysis problems that are actually not distributive, for instance pointer analysis or general constant propagation, into subproblems that are, in fact, distributive. Thus they can be efficiently solved using IFDS or IDE solvers, assuming some extra analysis code that then composes the results of individual distributive computations to the final analysis result. One goal of the optimization engine we seek to develop here will be to identify the potential for such a decomposition of analysis problems automatically.

4 CONCLUSION

We have presented the novel idea of enabling self-adaptive, self-optimizing static analyses, by developing a dedicated domain-specific language and multiple dedicated domain-specific intermediate representations (IR) not to express programs but to express the analysis itself. A novel low-level IR, in particular, is meant to represent how a particular analysis executes on a particular program. This enables just-in-time optimizations making use of both static-analysis and program properties. We have presented an overall solution architecture that has some resemblence to application-level just-in-time optimizers, and have highlighted the core challenges the community will face in developing a concrete solution.

REFERENCES

[1] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1371–1382.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI 2014*. ACM, 259–269.

[3] Eric Bodden, Laurie Hendren, and Ondrej Lhoták. 2007. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP 2007*. Springer-Verlag, 525–549.

[4] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices* 44, 10 (2009), 243–262.

[5] Peter F Haggar, James A Mickelson, and David Wendt. 2005. Single-instance class objects across multiple JVM processes in a real-time system. (Jan. 11 2005). US Patent 6,842,759.

[6] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *CAV 2016*. Springer, 422–430.

[7] John B Kam and Jeffrey D Ullman. 1977. Monotone data flow analysis frameworks. *Acta Informatica* 7, 3 (1977), 305–317.

[8] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. 2009. *Data flow analysis: theory and practice*. CRC Press.

[9] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. 2015. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis With Unbounded Access Paths. In *ASE 2015*. 619–629.

[10] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking Load-time Configuration Options. In *ASE 2014*. 445–456.

[11] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From datalog to flix: A declarative language for fixed points on lattices. In *PLDI 2016*. ACM, 194–208.

[12] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL 1995 (POPL '95)*. ACM, 49–61.

[13] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming* 58, 1-2 (2005), 206–263.

[14] Micha Sharir and Amir Pnueli. 1978. Two approaches to interprocedural data flow analysis. (1978).

[15] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *POPL 2011*. ACM, 17–30.

[16] Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDEal: Efficient and Precise Alias-aware Dataflow Analysis. In *OOPSLA/SPLASH 2017*. ACM Press. To appear.

[17] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA 2012*. ACM, 254–264.

[18] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical Virtual Method Call Resolution for Java. In *OOPSLA 2000*. ACM, 264–280.

[19] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java bytecode using the Soot framework: Is it feasible?. In *Compiler Construction (CC)*. Springer.

[20] Christian Wimmer, Michael Haupt, Michael L Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An approachable virtual machine for, and in, java. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 30.

[21] Xiao Xiao and Charles Zhang. Geometric encoding: forging the high performance context sensitive points-to analysis for Java. In *ISSTA 2011*. ACM, 188–198.

[22] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. 2014. LeakChecker: Practical static memory leak detection for managed languages. In *CGO*. ACM.