Evaluating the Experimental Complexity of Program Analysis

M2 level internship proposal, to be filled Fall 2024 or Spring 2025

Raphaël Monat & Julien Forget
Informal enquiries are welcome by email

1 Context – Static Program Analysis

One approach aiming at reducing the number of bugs is static program analysis through the framework of abstract interpretation [2]. Contrary to dynamic analyses such as fuzzing [9], the program is not executed but its source code is analyzed. Thanks to this approach, the analysis conservatively considers all possible execution paths of the program during the analysis, ensuring the absence of false negatives. In addition, the analyses are automatic: they do not require any user interaction to complete their task and they will be completed in a guaranteed finite time. These analyses can be seen as "push-button" as no expert knowledge is required to run them. This approach has been particularly successful to certify the absence of runtime errors in critical embedded C software. Astrée [3] has proved the absence of runtime errors in software of Airbus planes. More recently, Frama-C's static analysis has been used on the code of nuclear power plants [1].

However, the daily use of conservative static analyzers by non-experts remains a challenge. These tools can offer a wide range configuration options, where each will impact the performance-precision tradeoff of the analysis of a given program. Mansur et al. [7], Heo et al. [5] have looked into ways to automatically choose options to attain the highest precision when analyzing a program, given a resource envelope (CPU time, memory usage).

Here, we are broadly interested in the termination time of an analysis of a given program. Static analyzers are designed to terminate in finite time, but this property is not strong enough in practice. First, analyses that terminate in bounded time (such as a year) may not yield actionable results in time. Second, most static analyzers do not express their progress during an analysis, which results in an unfriendly black-box behavior.

2 Goal – Experimental Complexity of Program Analysis

The overall goal of this internship is to find ways around the usability barrier related to the termination of static analyzers, by being able to estimate the analysis time of a given program. This estimation will happen either offline (before the analysis), or online (during the analysis, similarly to a progress bar [6]). In the end, we plan to integrate the developed approaches within the Mopsa static analysis platform, a state-of-the-art static analyzer we are developping, and which won the "Software Systems" track of the academic Software Verification Competition [8].

In the case of context-sensitive analyses (working by virtual inlining of function calls) we are developing, we hypothesize that the complexity of analyzing a program depends on various factors, such as the number of programs loops and function calls, the maximum depth of these nested constructs, and the number of variables defined in the abstract state. The first step of the internship will be to confirm this hypothesis. Then, we will focus on finding measures of the complexity of a program's analysis, in a simplified setting where we consider a toy imperative language. The main goal of the internship is to introduce an "abstract cost" semantics, interpreting the cost of analyzing a given program through complexity formulas parameterized by the costs of different abstract domains used. In a way, the abstract cost semantics is an *analysis of the program analysis* itself [4]. If needed, we will consider additional, yet realistic, hypotheses on the convergence of widening in the initial program analysis to ease the definition of the abstract cost semantics. Finally, this method will be implemented in the Mopsa static analyzer. We will evaluate it on the Software Verification Competition (SV-Comp), where each program has to be analyzed as precisely as possible within 15 minutes of analysis time.

3 Requirements

Background in formal methods and programming language theory is a useful prerequisite. Familiarity with conservative static analysis, and in particular abstract interpretation is a plus. We expect the successful candidate to be motivated to improve experimental research tools such as Mopsa: knowledge of functional programming (such as OCaml) is required.

4 Logistics

The internship will take place in the SyCoMoRES team of Inria Lille & CRIStAL lab, which currently hosts 4 fellow PhD students and one ATER. Lille is a city close to Brussels, Paris & London, easily reachable by train, with a large student population and a number of cultural places & events. The lab has a very active equality and parity commission, organizing different kind of meetings, as well as outreach activities for high-schoolers. One of the advisors (Raphaël Monat) is an active member of this commission.

We plan to organize weekly research meetings during the internship. In addition, the intern will be able to attend monthly meetings with other Mopsa practitioners. This research project is part of ANR JCJC RAISIN. We will hold quarterly project meetings with Sophie Cerf (member of the project), who is a researcher at Inria with expertise in control theory for software systems. We have funding to cover an internship and, if the internship goes well for all involved parties, a PhD grant.

References

[1] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM*, (8), 2021. doi:10.1145/3470569.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977*, 1977. doi:10.1145/512950.512973.

[3] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In *ASIAN 2006*, 2006. doi:10.1007/978-3-540-77505-8_23.

[4] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. A^2I: abstract^2 interpretation. *Proc. ACM Program. Lang.*, (POPL), 2019. doi:10.1145/3290355.

[5] Kihong Heo, Hakjoo Oh, and Hongseok Yang. Resource-aware program analysis via online abstraction coarsening. In *ICSE 2019*, 2019. doi:10.1109/ICSE.2019.00027.

[6] Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. A progress bar for static analyzers. In *SAS 2014*, 2014. doi:10.1007/978-3-319-10936-7_12.

[7] Muhammad Numair Mansur, Benjamin Mariano, Maria Christakis, Jorge A. Navas, and Valentin Wüstholz. Automatically tailoring abstract interpretation to custom usage scenarios. In *CAV 2021*, 2021. doi:10.1007/978-3-030-81688-9_36.

[8] Raphaël Monat, Marco Milanese, Francesco Parolini, Jérôme Boillot, Abdelraouf Ouadjaout, and Antoine Miné. MOPSA-C: Improved verification for C programs, simple validation of correctness witnesses (competition contribution). In *Proc. TACAS*, 2024. doi:10.1007/978-3-031-57256-2_26.

[9] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. URL `https://www.fuzzingbook.org/`. Retrieved 2024-07-01 16:50:18+02:00.