Mopsa-C: Modular Domains and Relational Abstract Interpretation for C Programs (Competition Contribution)

Raphaël Monat[1]*(✉), Abdelraouf Ouadjaout[2], and Antoine Miné[3]

[1] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[2] Grenoble, France
[3] LIP6, Sorbonne Université, F-75005, Paris, France

Abstract. Mopsa is a multilanguage static analysis platform relying on abstract interpretation.
It is able to analyze C, Python, and programs mixing these two languages; we focus on the C analysis here. It provides a novel way to combine abstract domains, in order to offer extensibility and cooperation between them, which is especially beneficial when relational numerical domains are used. The analyses are currently flow-sensitive and fully context-sensitive. We focus only on proving programs to be correct, as our analyses are designed to be sound and terminating but not complete. We present our first participation to SV-Comp, where Mopsa earned a bronze medal in the *SoftwareSystems* category.

Keywords: Static analysis · Abstract interpretation · Competition on Software Verification · SV-Comp

1 Verification Approach: the Mopsa platform

Mopsa is an open-source static analysis platform relying on abstract interpretation [4]. The implementation of Mopsa aims at exploring new perspectives for the design of static analyzers. Mopsa has a triple objective:

– To allow developers to define abstract domains in a modular fashion – that is, as independently of each other as possible. In particular, this means that each abstract domain can easily be enabled or disabled to customize an analysis.
– To allow different abstract domains to cooperate and communicate in a relational way. Previous analyzers were able to combine domains in tree-shaped structures [5, Fig. 1]. Mopsa allows sharing between abstract domains, meaning schematically that the domains can be combined into an acyclic graph.
– To support the analysis of multiple languages while reusing existing abstractions. Mopsa is able to analyze C [16], Python [13], and multilanguage Python/C programs [14]. The Michelson smart contract language is being added [1]. Other safe analyzers, such as Astrée [5], Frama-C [6], Goblint [19], and TAJS [8] are specialized in analyzing a single language.

* Jury member
A. Ouadjaout—Unaffiliated.

© The Author(s) 2023

These aims are achieved through a dynamic expression rewriting mechanism, and a unified signature for abstract domains and iterators. Journault et al. [9] describe the core Mopsa principles, and Monat [12, Chapter 3] provides an in-depth introduction to Mopsa's design.

The C analysis which we rely on for this competition is based on the work of Ouadjaout and Miné [16]. The analysis works by induction on the syntax, is fully context- and flow-sensitive, and committed to be sound. It targets complete programs that have not been modified: Mopsa can be seamlessly integrated in standard build systems (such as `make`), it supports main functions with symbolic arguments, and it includes precise stubs for most of the standard C library. Our benchmarks analyses include, for instance, several tools from `coreutils`.

Mopsa is written in 50,000 lines of OCaml code [21], and relies on the Clang frontend to parse C programs. It relies on the Apron library [7] to handle relational numerical abstract domains.

2 Software Architecture: the SV-Comp driver

By default, the C analysis of Mopsa detects all the runtime errors that may happen in the analyzed program (NULL pointer dereferences, integer overflows, ...), while SV-Comp tasks focus on a specific property at a time (reachability of a function, validity of memory accesses, ...). We thus created an SV-Comp specific driver. It takes as input the task description (program, property, data model). It runs increasingly precise C analyses defined in Mopsa until the property of interest is proved or the most precise analysis is reached. Each analysis result is postprocessed by the driver to check if the property is proved.

An analysis configuration defines the set of domains used, as well as their parameters allowing modifications of the precision-efficiency ratio. The four increasingly precise configurations we use are the following:

– Conf. 1 is the base analysis relying on intervals and cells (a field-sensitive, low-level memory abstraction able to handle type-puning, pointer casts, C unions, . . .) [11]. Global structures having up to 5 fields are precisely initialized.
– Conf. 2 additionally enables the string length domain [10], which precisely tracks the position of the first 0 in byte arrays. Static struct initialization is done precisely for structures having up to 50 fields.
– Conf. 3 adds a polyhedra abstract domain. This includes tracking numerical relations between string lengths and scalar variables.[4] It relies on a static packing heuristic [5] to achieve a good precision-scalability tradeoff.
– Conf. 4 adds a congruence abstract domain, delayed widenings, and widening with thresholds.

A schematic representation of the domains used in these analyses is shown in Figure 1. The SV-Comp driver is written in 250 lines of Python code.

[4] In this case, Mopsa's ability to share abstract domain comes in handy. With a tree, we would have to "linearize" the domains and put either cells or string length on top of the other. This makes reduction more difficult (e.g., Astrée uses a global reduction system on the whole tree, while we can use local reductions between two domains).

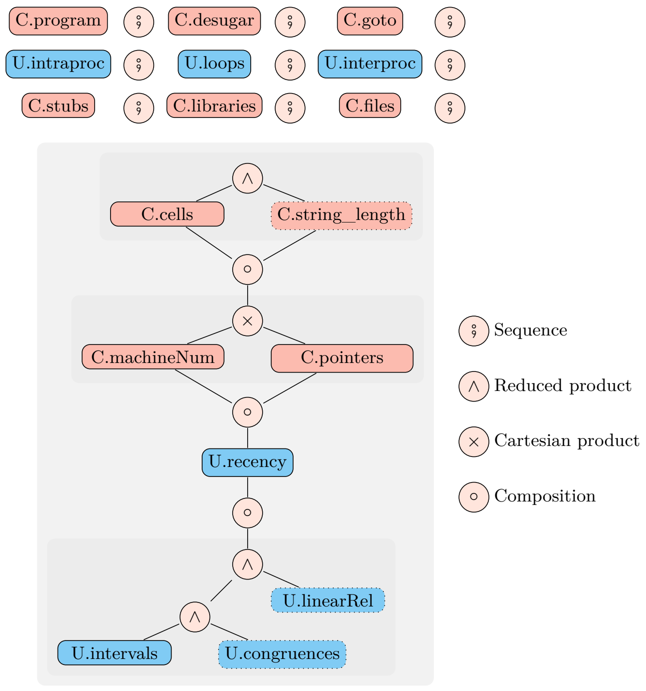

Fig. 1. Configurations for Mopsa-C analyses used in SV-Comp. Dotted rectangles indicate optionally enabled domains. "U.*" domains are shared between the analysis of different languages, while the others are C-specific. The sequence operator lets the domain on the left handle the analysis of a given statement: if it cannot, the analysis continues with the domain on the right. The composition operator allows multiple domains to share the same underlying domain. Products let both domains analyze the given statement. In the case of a reduced product, a reduction operation is applied after the analysis of a statement.

Conf.	Tasks proved correct		Tasks yielding timeout	
1	5695		279	
2	6433	(+738)	365	(+86)
3	6885	(+452)	1844	(+1479)
4	6909	(+24)	2009	(+165)

Fig. 2. Results of the increasingly precise analyses (21220 tasks in total, 12636 correctness tasks). Conf. 2 is able to prove 738 tasks correct in addition to the 5695 proved by conf. 1, although 86 tasks reach the resource limits when analyzed by conf. 1 and 2. Mopsa yields unknown in the analysis of the other tasks.

3 Strengths and Weaknesses

Mopsa participated in all categories targeting reachability, memory safety and overflow properties: *ReachSafety, MemSafety, NoOverflows* and *SoftwareSys-*

tems. It did not compete in the datarace and termination categories. The competition report [2] details all results.

Mopsa relies on over-approximations to guarantee soundness and termination of its analyses. As such, Mopsa scales well on SV-Comp benchmarks: the successive analyses described in Section 1 yield a result within the allocated resources in 91% of the tasks (and 98.5% of the cases for our cheapest analysis). We show the detailed precision benefits of each analysis for the benchmarks in Figure 2. Thanks to Mopsa's scalability and commitment to soundness, we have been able to discover and fix defects within SV-Comp benchmarks which were not discovered by previous tools. In particular, we fixed 164 task definitions, as well as 23 programs with unintended issues in their source code.[5] Mopsa is especially competitive in the *SoftwareSystems* category, focusing on verifying real software systems: it ranked third for our first participation.

Our approach is scalable but not complete: we can only prove programs correct. In other cases, we cannot decide if the issues we found are real bugs or false alarms: we return "unknown" in all these cases to avoid yielding incorrect results. Thus, we can only obtain points on correctness verification tasks, which represents around 58% of the current tasks. Our future work includes finding approaches to exhibit real counterexamples when they exist.

In addition, our analyses are not precise enough for some small but intricate benchmarks (for exemple, on arrays). In particular, the current version of Mopsa does not support partitioning the abstract state into different ones to improve its precision. We plan to add this classic feature for SV-Comp's next edition. For an over-approximating analyzer, Mopsa is nevertheless quite precise: Mopsa is able to prove around 8% more tasks than Goblint [19, 20] (the leading state-of-the-art abstract interpreter running in SV-Comp).

Finally, the SV-Comp driver we built does not extract precise witnesses from the analyses. Indeed, the case of invariant generation for loops defined in functions called in different contexts seems open for now: Saan [18] observed that complex, interprocedural witnesses do not help the witness verifiers. However, the trivial correctness witnesses we generate are validated in 96.4% of the cases.

4 Software Project and Contributors

Mopsa is currently available on Gitlab[17], and released under an open-source license (GNU LGPL v3). Mopsa was originally developed at LIP6, Sorbonne Université following an ERC Consolidator Grant award to Antoine Miné. Mopsa is now developed in other places, including Inria, Airbus, and Nomadic Labs. We thank Matthieu Journault for being one of the initial contributors to Mopsa. This first participation to SV-Comp has spurred a lot of interesting discussions within our development team, and lead to 20 bugfixes and new features.

[5] We also added contributed to the benchmarks used in SV-Comp, by adding tasks to check overflows from the Juliet Benchmarks (6156 new tasks); and reviewing 12 merge requests from the community.

Data-Availability Statement The exact version of Mopsa that participated in SV-Comp 2023, and our specific driver are available as a Zenodo archive [15]. A global tool archive is also available [3].

Acknowledgements. We thank Simmo Saan for his precious advice on how to start integrating our tool within SV-Comp.

References

1. Bau, G., Miné, A., Botbol, V., Bouaziz, M.: Abstract interpretation of michelson smart-contracts. In: ACM SOAP, pp. 36–43 (2022)
2. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2), LNCS , Springer (2023)
3. Beyer, D.: Verifiers and validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023) (2023), https://doi.org/10.5281/zenodo.7627829
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
5. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the Astrée static analyzer. In: ASIAN, pp. 272–300 (2006)
6. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - A software analysis perspective. In: SEFM, pp. 233–247 (2012)
7. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV, pp. 661–667, Springer (2009)
8. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: SAS, pp. 238–255 (2009)
9. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, pp. 1–18 (2019)
10. Journault, M., Miné, A., Ouadjaout, A.: Modular static analysis of string manipulations in C programs. In: SAS, pp. 243–262 (2018)
11. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: LCTES (2006)
12. Monat, R.: Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries. Ph.D. thesis, Sorbonne Université, France (2021)
13. Monat, R., Ouadjaout, A., Miné, A.: Static type analysis by abstract interpretation of python programs. In: ECOOP, pp. 1–29 (2020)
14. Monat, R., Ouadjaout, A., Miné, A.: A multilanguage static analysis of python programs with native C extensions. In: SAS, pp. 323–345 (2021)
15. Monat, R., Ouadjaout, A., Miné, A.: Mopsa-C: Modular Domains and Relational Abstract Interpretation for C Programs (Artefact) (Dec 2022), https://doi.org/10.5281/zenodo.7467136
16. Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: SAS, pp. 223–247 (2020)
17. Ouadjaout, A., Monat, R., Miné, A., Journault, M.: Mopsa (2022), URL https://gitlab.com/mopsa/mopsa-analyzer
18. Saan, S.: Witness generation for data-flow analysis. https://comserv.cs.ut.ee/home/files/saan_computerscience_2020.pdf (2020)

19. Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints - (competition contribution). In: TACAS (2021)
20. Saan, S., Schwarz, M., Erhard, J., Pietsch, M., Seidl, H., Tilscher, S., Vojdani, V.: Goblint: Autotuning thread-modular abstract interpretation (competition contribution). In: Proc. TACAS (2), LNCS , Springer (2023)
21. The OCaml Developers: Ocaml (2020), URL https://github.com/ocaml/ocaml

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.