Mopsa-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution)

Raphaël Monat[1(✉)⋆], Marco Milanese[2], Francesco Parolini[2], Jérôme Boillot[3], Abdelraouf Ouadjaout[4], and Antoine Miné[2]

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

Abstract. We present advances we brought to Mopsa for SV-Comp 2024. We significantly improved the precision of our verifier in the presence of dynamic memory allocation, library calls such as `memset`, `goto`-based loops, and integer abstractions. We introduced a witness validator for correctness witnesses. Thanks to these improvements, Mopsa won SV-Comp's *SoftwareSystems* category by a large margin, scoring 2.5 times more points than the silver medalist, Bubaak-SpLit.

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 [6]. The implementation of Mopsa aims at exploring new perspectives for the design of static analyzers. Journault et al. [8] 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]; it proceeds by induction on the syntax, is fully context- and flow-sensitive, and committed to be sound. This is the second time Mopsa participates in SV-Comp [15]. We have brought precision improvements, described below; they have proved decisive for the *SoftwareSystems* category.

Dynamic memory allocation precision improvements. Mopsa relies on the recency abstraction [1] to handle dynamic allocation. For each allocation site, this abstraction keeps the last allocated block separated from the others, the latter being summarized into a single, weak memory block. Allocation sites are customizable [14], they are usually based on a program location. However, this summarization can be detrimental to precision. We implemented an alternative abstraction that keeps memory blocks separated during loop unrolling. This

⋆ Jury member

enhancement, combined with targeted loop unrolling helped us verify more tasks, including 246 from the *uthash* categories. Specifically, we proved correct half the tasks of *uthash-NoOverflow* category, which are out of reach of the other verifiers.

Integer abstractions. Mopsa only supported convex representations of integer sets, such as intervals. As such, it was impossible to precisely represent cases where $x \in [-10, 10]$ and $x \neq 0$. We have resolved this issue by adding an excluded set domain, which tracks a set of values a given variable cannot take. We have also implemented the symbolic rewriting domain of Boillot and Feret [4], which simplifies arithmetic expressions with overflows into simpler ones. This new implementation has been written in 1,200 lines of OCaml code.

Improved precision for goto-based loops. Since the analyzer iterates on the syntax of the program, `goto` statements require the usage of flows tokens [12] and a special fixpoint iteration scheme. We added support for a decreasing iteration pass, which allows to recover some precision after the generalization performed by the widening operator. In addition, we added a syntactical loop rewriting pass which turns few special goto patterns into equivalent while loops which are analyzed more precisely.

Precise stub initialization. Ouadjaout and Miné [16] implemented a stub language and its interpretation for the C standard library in Mopsa. Contiguous region initialization through functions such as `memset` were not handled precisely by our implementation of the cells domain [11], mainly to be scalable. We improved the domain to handle region initialization up to a given bound, and `NULL` pointer synthesis from a contiguous block of 0 bytes.

Other improvements. Some SV-Comp programs have specific symbolic argument initialization performed by client code, with variable parameters on the maximal size of all symbolic arguments. We have thus extended Mopsa to handle a wide range of parameters for symbolic argument initialization, matching those found in SV-Comp programs. We also rely on the flambda optimizer for OCaml, which brings more than a 15% performance improvements.

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, while SV-Comp tasks focus on a specific property at a time. We thus rely on 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 (or the resources are exhausted). Each analysis result is postprocessed by the driver to check if the property is proved.

An analysis configuration defines the set of domains used, and their parameters allowing modifications of the precision-efficiency ratio. A breakdown of the results is shown in Fig. 1. This year, we use five configurations. Conf. 1 relies on intervals and cells [11]. Conf. 2 additionally enables the string length domain [9], the excluded powerset domain, and congruences. It performs decreasing iterations for `goto` statements, unrolls the first 10 iterations of loops, enables the

Max. Conf.	Tasks proved correct		Tasks yielding timeout	
1	6995		368	
2	7775	(+780)	717	(+349)
3	8197	(+422)	2954	(+2237)
4	8257	(+60)	3527	(+573)
5	8400	(+143)	9532	(+6005)

Fig. 1. Max. Conf. i represents the sequence of increasingly precise analyses from Conf. 1 up to Conf. i. Max. Conf. 2 is able to prove 780 tasks correct in addition to the 6995 proved by conf. 1, although 717 tasks reach the resource limits when analyzed by Conf. 1 and 2 (349 more than by Conf. 1 alone). There are 25885 tasks in total, and 17851 correctness tasks. Mopsa can only prove program correctness for now (68% of the tasks); it yields "unknown" when unable to prove a program correct.

enhanced memory allocation abstraction, and the more precise evaluation of stubs. Conf. 3 adds a polyhedra abstract domain, relying on a static packing to scale [7]. This includes tracking numerical relations between string lengths and scalar variables. A pointer sentinel domain is added to symbolically track the position of the first `NULL` value of a pointer array. Decreasing iterations are also enabled `for`/`while` loops, and the first 15 iterations of loops are unrolled. Conf. 4 adds the symbolic rewriting domain of Boillot and Feret [4]. Loop unrolling is extended to 60 iterations. Conf. 5 performs a fully relational analysis of the analyzed program without packing.

Witness Validation. We extended our driver to support the witness validation phase of SV-Comp: we inject loop invariants of a witness, encoded as assertions into the original program. We then check that this patched program is correct. This approach is similar to Metaval's [3], but we used the new YAML format. The work of Saan et al. [22] is more involved: it leverages the witness to guide their analysis and yields precision improvements, compared to their bare analysis.

3 Strengths and Weaknesses

Mopsa participated in the following categories, targeting C programs: *ReachSafety, MemSafety, NoOverflows* and *SoftwareSystems*. It did not compete in the termination category and cannot precisely analyze concurrency-related verification tasks. The highlight of this year's participation is Mopsa's gold medal in the *SoftwareSystems* track, focusing on verifying real software systems. Mopsa scored 2.5 times more points than the second tool, Bubaak-SpLit [5]. Figure 2 breaks down the results of Mopsa in the subcategories of the *SoftwareSystems* track, highlighting our progress, and the best results obtained by this year's verifiers. An overview of results can be found in the competition report [2].

Strengths. Mopsa is quite scalable: our cheapest configuration is able to analyze a given program within the allocated resource budget in 98.6% of the cases. In addition, Mopsa is the only verifier of 2023 and 2024 able to gain points in the DDLL category, corresponding to large instances of instrumented Linux drivers.

Category	Prop.	|tasks|	Mopsa'23	Mopsa'24	Best score (2024)	
AWS	R	197	32	36	137	Symbiotic
coreutils	M	140	0	0	0	_
coreutils	N	30	0	4	4	Mopsa
BusyBox	N	54	4	8	8	Mopsa
DDL	R	2442	3174	3476	3476	Mopsa
DDLL	R	8	10	14	14	Mopsa
DDL	M	141	0	8	71	Bubaak-SpLit
other	R	22	0	10	10	Mopsa
other	M	34	0	12	12	Mopsa
uthash	R	138	0	192	228	Bubaak*, Symbiotic
uthash	M	138	0	96	204	Bubaak*, Symbiotic
uthash	N	114	0	204	204	Mopsa

Fig. 2. Mopsa's improvements for subcategories of the *SoftwareSystems* track. Property is either ***R**eachSafety*, ***M**emSafety* or ***N**oOverflow*. The last three columns show the score of Mopsa submitted last year, this year, and the best score reached by a verifier.

Mopsa is committed to being sound. Thanks to this, we have been able to fix 20 mislabeled verdicts this year, mainly in the DDL category (*DeviceDriversLinux*).

Weaknesses. Mopsa can only prove programs correct for now, and is currently unable to provide counterexamples otherwise. We plan to leverage the recent work of Milanese and Miné [10] to address this issue. Our SV-Comp driver currently tries a fixed sequence of increasingly precise configurations. We plan to reuse information between the different analyses of the sequence, and automatically adapt the options of Mopsa to the analyzed program (similar to Goblint's autotuning [21]). Our analysis is not competitive enough in the tracks besides *SoftwareSystems*: we plan to add new array abstractions as well as a partitioning mechanism. We also noted that Mopsa is imprecise on `longjmp`, following the addition of recent benchmarks from Schwarz et al. [23] to SV-Comp.

Methodology. We finish this section by explaining how we worked to improve Mopsa this year. We focused on the most important subcategories of *SoftwareSystems*. We encountered a few runtime errors in our analysis: we used automated testcase reduction [18] to pinpoint these issues and fix them. We investigated several timeouts in the *DeviceDriversLinux-Large* (DDLL) category, by using standard profiling tools (such as `perf`), but also by profiling which parts of a given program took long to analyze through custom plugins. The rest of the work consisted in performing manual inspection of some tasks to see how we could improve precision. We started by choosing tasks solved by competing tools relying on similar approaches, starting from Goblint [20, 21, 19].

4 Software Project and Contributors

Mopsa is available on Gitlab [17], and released under an GNU LGPL v3 license. Mopsa was originally developed at LIP6, Sorbonne Université following an ERC Consolidator Grant award to Antoine Miné. Mopsa is now additionally developed in other places, including Inria, ENS Airbus, and Nomadic Labs. The people who improved Mopsa for SV-Comp 2024 are the authors of this paper.

Data-Availability Statement. The exact version of Mopsa and the driver that participated in SV-Comp 2024 are available as a Zenodo archive [13].

Bibliography

[1] Balakrishnan, G., Reps, T.W.: Recency-abstraction for heap-allocated storage. In: SAS, Lecture Notes in Computer Science, vol. 4134, pp. 221–239, Springer (2006)

[2] Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS, LNCS , Springer (2024)

[3] Beyer, D., Spiessl, M.: Metaval: Witness validation via verification. In: CAV (2), Lecture Notes in Computer Science, vol. 12225, pp. 165–177, Springer (2020)

[4] Boillot, J., Feret, J.: Symbolic transformation of expressions in modular arithmetic. In: SAS, Lecture Notes in Computer Science, vol. 14284, pp. 84–113, Springer (2023)

[5] Chalupa, M., Richter, C.: Bubaak-SpLit: Split what you cannot verify (competition contribution). In: Proc. TACAS, LNCS , Springer (2024)

[6] 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)

[7] 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)

[8] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, pp. 1–18 (2019)

[9] Journault, M., Miné, A., Ouadjaout, A.: Modular static analysis of string manipulations in C programs. In: SAS, pp. 243–262 (2018)

[10] Milanese, M., Miné, A.: Generation of Violation Witnesses by Under-Approximating Abstract Interpretation. In: VMCAI, Springer (2024)

[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., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: Mopsa at sv-comp 2024 (Nov 2023), `https://doi.org/10.5281/zenodo.10198570`

[14] Monat, R., Ouadjaout, A., Miné, A.: Value and allocation sensitivity in static python analyses. In: SOAP@PLDI, pp. 8–13, ACM (2020)

[15] Monat, R., Ouadjaout, A., Miné, A.: Mopsa-c: Modular domains and relational abstract interpretation for C programs (competition contribution). In: TACAS (2), Lecture Notes in Computer Science, vol. 13994, pp. 565–570, Springer (2023)

[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] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: PLDI, pp. 335–346, ACM (2012)
[19] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint: Abstract interpretation for memory safety and termination (competition contribution). In: Proc. TACAS, LNCS , Springer (2024)
[20] 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)
[21] 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)
[22] Saan, S., Schwarz, M., Erhard, J., Seidl, H., Tilscher, S., Vojdani, V.: Correctness witness validation by abstract interpretation. In: VCMAI, LNCS , Springer (2024)
[23] Schwarz, M., Erhard, J., Vojdani, V., Saan, S., Seidl, H.: When long jumps fall short: Control-flow tracking and misuse detection for non-local jumps in C. In: SOAP@PLDI, pp. 20–26, ACM (2023)