Mopsa-C with Trace Partitioning and Autosuggestions (Competition Contribution)

Raphaël Monat[1 (✉)⋆], Abdelraouf Ouadjaout[2], 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

Abstract. We present advances we brought to Mopsa for SV-Comp 2025. Most notably, Mopsa now supports bounded trace partitioning, constant widening with thresholds, and can check that all memory has been correctly deallocated. Further, Mopsa now integrates a sound support of bitfields. While Mopsa at SV-Comp previously relied on a fixed, homogeneous set of configurations to verify tasks, it can now automatically leverage semantic information from a previous analysis to trigger heuristic precision improvements in further analyses. With these improvements, Mopsa wins a silver medal in the *SoftwareSystems* category and ranks fifth in the *NoOverflows* 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 [9]. The implementation of Mopsa aims at exploring new perspectives for the design of static analyzers. Journault et al. [13] describe the core of Mopsa principles, and Monat [22, 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 [27]; it proceeds by induction on the syntax, is fully context- and flow-sensitive, and committed to be sound. This is the third time Mopsa participates in SV-Comp [24, 23]. We have brought several enhancements, including major precision improvements, described below.

Trace partitioning. Mopsa now supports a variant of trace partitioning [16]. Trace partitioning keeps some abstract states separate (depending on the analysis trace) to improve precision. In our implementation, we keep a small, bounded abstract trace to separate abstract states while maintaining full analysis coverage. The abstract trace consists in the k latest trace markers. Currently, trace markers correspond to control conditions (`if`, `switch`, different `return` locations), and case disjunctions when handling C stubs [27].

⋆ Jury member

Widening with constant thresholds. Mopsa relies on the traditional abstract interpretation use of widening [9] to enforce finite convergence when analyzing loops. However these widening operators may generalize some constraints too quickly, which can be difficult to recover from. To address this issue, Mopsa now supports the standard widening with thresholds [4]. It is implemented as a plug-in, which observes the analyzer and performs some constant propagation to decide relevant thresholds for each variable. This approach greatly improves precision when analyzing loops guarded by (in)equalities.

Memory deallocation check. Mopsa can prove that programs have successfully deallocated all memory. During the end of the analysis of a program, it queries the recency abstraction [2] to ensure that all memory dynamically allocated through `malloc` and other glibc functions has been properly deallocated. This allows us to support the *MemCleanup* property of SV-Comp (especially in the corresponding `uthash` subcategory of *SoftwareSystems*).

Sound bitfield support. The low-level C memory representation of cells [20] we use works at the byte level, making bitfield reasoning hard. We have thus implemented a domain translating bitfield operations into equivalent byte-level operations with bitmasks. This approach is currently sound, although imprecise. The precision could be improved through new numeric abstractions in the future.

Other improvements. Mopsa leverages relational numeric domain through Apron's interface [12], relying on static packing [4] to remain scalable. In order to improve precision for small and intricate programs, the last configuration used in our SV-Comp driver (Section 2) does not rely on packing (i.e., all variables are used in the same polyhedron). Noticing performance improvements in this case, we decided to rely on the PPLite polyhedra implementation [3]. Mopsa does not support the analysis of recursive functions. We added support to unroll recursive functions of bounded depth.

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 sequentially tries 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 control of the precision-efficiency ratio. A breakdown of the results is shown in Fig. 1. Similarly to last year [23], we use five configurations. Confs. 1 and 3 are unchanged from last year. Conf. 2 now only unrolls the first 2 iterations of loops (down from 10 last year – but the precision suggestion hook mentioned below can override this parameter). Conf. 4 adds the bounded trace partitioning, where up to 7 trace markers can be kept. Conf. 5 also enables bounded trace partitioning (with up to 10 trace markers), and relies on PPLite without static packing for best precision.

Max. Conf.	Tasks proved correct		Tasks reaching 900s timeout	
1	7002		389	
2	7743	(+741)	970	(+581)
3	8489	(+746)	3377	(+2407)
4	8660	(+171)	5378	(+2001)
5	8933	(+273)	8440	(+3062)

Fig. 1. Results of our sequential portfolio at SV-Comp 2025. Max. Conf. i represents the sequence of increasingly precise analyses from Conf. 1 up to Conf. i. Max. Conf. 2 is able to prove 741 tasks correct in addition to the 7002 proved by Conf. 1, although 970 tasks reach the resource limits when analyzed by Conf. 1 and 2 (581 more than by Conf. 1 alone). There are 33592 tasks in total, including 22356 correctness tasks. Mopsa can only prove program correctness for now; it yields "unknown" when unable to prove a program correct.

Heuristic Autosuggestions. An important improvement is that the set of configurations is not fixed and uniform for all programs anymore. Each analysis can suggest enabling options that will improve the precision of further analyses. These options are decided by a plug-in observing the analysis of Mopsa. It currently supports three semantic heuristics:

Bounded recursion unrolling. Upon detection of a call to a recursive function `f` having parameter `n`, if `n` lies in interval $[0, \texttt{hi}]$, and $\texttt{hi} < 1000$, further analyses will inline recursive functions up to bound `hi`.[3]

Loop unrolling for precise allocations. If the program contains a loop, for which we can semantically infer that (i) there are less than 30 iterations and that (ii) iterations perform allocations, further analyses will unroll this loop to keep the allocated memory blocks distinct and enhance precision.

Single loop unrolling. If the program semantically reaches a single loop, for which we can infer an upper bound on the number of iterations, further analyses will fully unroll this loop.

3 Strengths and Weaknesses

Mopsa participated in the following categories, targeting C programs: *ReachSafety, MemSafety, NoOverflows* and *SoftwareSystems*. An overview of results can be found in the competition report [5]. Figure 2 highlights the progress made by Mopsa in specific subcategories, thanks to the improvements brought for this year's edition. Note that in two subcategories, the score of Mopsa decreased due to our now-sound bitfield encoding.

Strengths. Mopsa is quite scalable: our cheapest configuration is able to analyze a given program within the allocated resource budget in 98.8% of the cases. Thanks to this scalability, Mopsa is particularly competitive in the *SoftwareSystems* track, focusing on verifying real software systems. This year, Mopsa ranked

[3] In other cases, Mopsa relies on the function's prototype to return the top value, and assumes the recursive function has no side-effects. To keep this approach sound, our SV-Comp driver thus returns `unknown` whenever a recursive function has been encountered during the analysis.

Category	Prop.	|tasks|	Mopsa'24	Mopsa'25	Best score, verifier (2025)	
Hardness	R	4012	432	518	7426	SVF-SVC [17]
Heap	R	240	190	226	314	PredatorHP [29]
Loops	R	774	298	376	1031	AISE [34, 14]
Recursive	R	160	12	60	150	UTaipan [10]
Heap	M	247	40	154	331	PredatorHP [29]
Juliet	M	3271	2224	2530	4709	CPAchecker [1]
LinkedLists	M	134	58	96	220	PredatorHP [29]
Main	N	1989	1920	2138	2756	UAutomizer [11]
AWS	R	341	36	76	326	Bubaak [6, 8]
DDL	R	2420	3476	3602	3602	Mopsa
uthash	M	192	96	108	246	Bubaak* [6, 8, 7]
uthash	N	162	204	300	300	Mopsa

Fig. 2. Mopsa's improvements for selected subcategories of the *ReachSafety, MemSafety, NoOverflows* and *SoftwareSystems* tracks, comparing the scores reached at SV-Comp 2024 and 2025. 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.

second with 2164 points, closely trailing CPAchecker [1] with 2238 points. It is the best verifier in the *DeviceDriversLinux-ReachSafety* and the *uthash-NoOverflows* subcategories. In *uthash-NoOverflows* there are only two verifiers able to score points; the second is UAutomizer [11], with 6 points. The second strength of Mopsa lies in the *NoOverflows* track, where it ranked fifth, with results near those of Goblint [30], which is the first abstract-interpretation based verifier to enter the competition.

Weaknesses. Mopsa can only prove programs correct for now, and is currently unable to provide counterexamples otherwise. We plan to leverage the recent works of Milanese and Miné [18, 19] to address this issue. Mopsa does not support the termination property, and cannot precisely analyze concurrency-related verification tasks, but we could leverage previous abstract interpretation work targeting those properties [32, 31, 21, 33]. Our SV-Comp driver currently tries a sequence of increasingly precise configurations: this approach is not efficient, we are planning to develop techniques deciding what would be the best configuration to analyze a given program, following the works of Oh et al. [26], Mansur et al. [15], Wang et al. [35].

4 Software Project and Contributors

Mopsa is available on Gitlab [28], 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 2025 are the authors of this paper.

Data-Availability Statement. The exact version of Mopsa and the driver that participated in SV-Comp 2025 are available as a Zenodo archive [25].

Bibliography

[1] Baier, D., Beyer, D., Chien, P., Jankola, M., Kettl, M., Lee, N., Lemberger, T., Rosenfeld, M.L., Spiessl, M., Wachowitz, H., Wendler, P.: Cpachecker 2.3 with strategy selection - (competition contribution). In: TACAS (3), Lecture Notes in Computer Science, vol. 14572, pp. 359–364, Springer (2024)

[2] 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)

[3] Becchi, A., Zaffanella, E.: Pplite: Zero-overhead encoding of NNC polyhedra. Inf. Comput. **275**, 104620 (2020)

[4] Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. Foundations and Trends in Programming Languages pp. 71–190 (2015)

[5] Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS, LNCS, Springer (2025)

[6] Chalupa, M., Henzinger, T.A.: Bubaak: Runtime monitoring of program verifiers - (competition contribution). In: TACAS (2), Lecture Notes in Computer Science, vol. 13994, pp. 535–540, Springer (2023)

[7] Chalupa, M., Richter, C.: Bubaak-split: Split what you cannot verify (competition contribution). In: TACAS (3), Lecture Notes in Computer Science, vol. 14572, pp. 353–358, Springer (2024)

[8] Chalupa, M., Richter, C.: Bubaak: Dynamic cooperative verification (competition contribution). In: Proc. TACAS, LNCS, Springer (2025)

[9] 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)

[10] Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate taipan and race detection in ultimate - (competition contribution). In: TACAS (2), Lecture Notes in Computer Science, vol. 13994, pp. 582–587, Springer (2023)

[11] Heizmann, M., Barth, M., Dietsch, D., Fichtner, L., Hoenicke, J., Klumpp, D., Naouar, M., Schindler, T., Schüssele, F., Podelski, A.: Ultimate automizer and the commuhash normal form - (competition contribution). In: TACAS (2), Lecture Notes in Computer Science, vol. 13994, pp. 577–581, Springer (2023)

[12] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV, pp. 661–667, Springer (2009)

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

[14] Lin, Y., Chen, Z., Wang, J.: AISE v2.0: Combining loop transformations (competition contribution). In: Proc. TACAS, LNCS, Springer (2025)

[15] Mansur, M.N., Mariano, B., Christakis, M., Navas, J.A., Wüstholz, V.: Automatically tailoring abstract interpretation to custom usage scenarios.

In: CAV (2), Lecture Notes in Computer Science, vol. 12760, pp. 777–800, Springer (2021)
[16] Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: ESOP, Lecture Notes in Computer Science, vol. 3444, pp. 5–20, Springer (2005)
[17] McGowan, C., Richards, M., Sui, Y.: SVF-SVC: Software verification using SVF (competition contribution). In: Proc. TACAS, LNCS, Springer (2025)
[18] Milanese, M., Miné, A.: Generation of Violation Witnesses by Under-Approximating Abstract Interpretation. In: VMCAI, Springer (2024)
[19] Milanese, M., Miné, A.: Under-approximating memory abstractions. In: Proc. of the 31th International Static Analysis Symposium (SAS'24), Lecture Notes in Computer Science (LNCS), vol. 14995, Springer (Oct 2024), http://www-apr.lip6.fr/~mine/publi/article-milanese-al-sas24.pdf
[20] Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: LCTES (2006)
[21] Miné, A.: Relational thread-modular static value analysis by abstract interpretation. In: VMCAI, Lecture Notes in Computer Science, vol. 8318, pp. 39–58, Springer (2014)
[22] Monat, R.: Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries. Ph.D. thesis, Sorbonne Université, France (2021)
[23] Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: Mopsa-c: Improved verification for C programs, simple validation of correctness witnesses (competition contribution). In: TACAS (3), Lecture Notes in Computer Science, vol. 14572, pp. 387–392, Springer (2024)
[24] 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)
[25] Monat, R., Ouadjaout, A., Miné, A.: Mopsa at sv-comp 2025 (Nov 2024), https://doi.org/10.5281/zenodo.14208644
[26] Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective context-sensitivity guided by impact pre-analysis. In: PLDI, pp. 475–484, ACM (2014)
[27] Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: SAS, pp. 223–247 (2020)
[28] Ouadjaout, A., Monat, R., Miné, A., Journault, M.: Mopsa (2022), URL https://gitlab.com/mopsa/mopsa-analyzer
[29] Peringer, P., Soková, V., Vojnar, T.: Predatorhp revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: TACAS (2), Lecture Notes in Computer Science, vol. 12079, pp. 408–412, Springer (2020)
[30] 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)

[31] Urban, C.: Function: An abstract domain functor for termination - (competition contribution). In: TACAS, Lecture Notes in Computer Science, vol. 9035, pp. 464–466, Springer (2015)
[32] Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: SAS, Lecture Notes in Computer Science, vol. 8723, pp. 302–318, Springer (2014)
[33] Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the goblint approach. In: ASE, pp. 391–402, ACM (2016)
[34] Wang, Z., Chen, Z.: AISE: A symbolic verifier by synergizing abstract interpretation and symbolic execution (competition contribution). In: TACAS (3), Lecture Notes in Computer Science, vol. 14572, pp. 347–352, Springer (2024)
[35] Wang, Z., Yang, L., Chen, M., Bu, Y., Li, Z., Wang, Q., Qin, S., Yi, X., Yin, J.: Parf: Adaptive parameter refining for abstract interpretation. In: ASE, pp. 1082–1093 (2024)