The Static Analyzer Frama-C in SV-COMP (Competition Contribution)

Dirk Beyer and Martin Spiessl

LMU Munich, Munich, Germany

Abstract. Frama-C is a well-known platform for source-code analysis of programs written in C. It can be extended via its plug-in architecture by various analysis backends and features an extensive annotation language called ACSL. So far it was hard to compare Frama-C to other software verifiers. Our competition participation contributes an adapter named Frama-C-SV, which makes it possible to evaluate Frama-C against other software verifiers. The adapter transforms standard verification tasks (from the well-known SV-Benchmarks collection) in a way that can be understood by Frama-C and produces a verification witness as output. While Frama-C provides many different analyses, we focus on the Evolved Value Analysis (EVA), which uses a combination of different domains to over-approximate the behavior of the analyzed program.

Keywords: Software verification · Program analysis · Formal methods · Competition on Software Verification · Comparative Evaluation · SV-COMP · Frama-C

1 Approach

This competition contribution is based on Frama-C [12], a program-analysis platform for C programs. The purpose of the participation in the comparative evaluation SV-COMP is to show the strengths of Frama-C when applied to the problem of verifying C programs from the SV-Benchmarks [4] collection of verification tasks.

2 Architecture

Although Frama-C has a large configuration space, it does not support standard specifications as used in SV-COMP, and it does not produce verification witnesses as default. In order to overcome this obstacle we implemented an adapter for Frama-C using input and output transformers, and the adaption architecture is illustrated in Fig. 1. In the following, we describe the artifacts and actors of the participating verifier: in Sect. 2.1 we describe all the components that are developed as part of the adapter, while in Sect. 2.2 we describe in more detail how the used EVA analysis of Frama-C works.

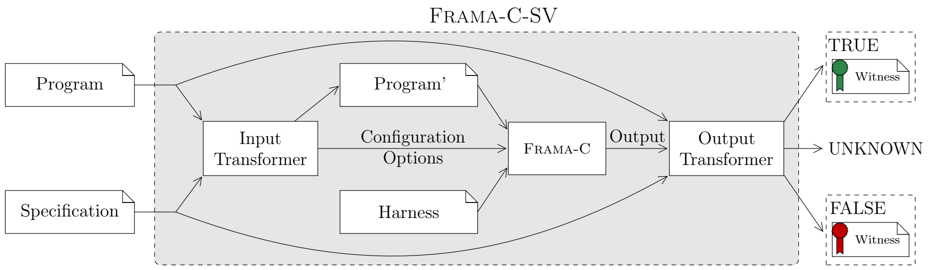

Fig. 1: Architecture of FRAMA-C-SV: the inputs and outputs of FRAMA-C are translated to interface with the established standards as used by SV-COMP; the components that are necessary to adapt FRAMA-C for comparison with other verifiers amount to 678 lines of code mostly written in Python

2.1 FRAMA-C-SV

Input Transformer. The input transformer takes the program p and specification s and creates a new program p' in which the specification s has been expressed as FRAMA-C-specific annotations. FRAMA-C uses ACSL [1] as language to specify annotations. The input transformer also selects configuration parameters for FRAMA-C that are best suited for the verification task. Currently we encode reachability tasks into signed integer overflows by adding an artificial overflow to the body of the function `reach_error`. This works well in practice and is also sound, since if there were any other overflows, the task would contain undefined behavior and would not be a valid reachability task in the first place.

Configuration Options. Depending on the input program and specification, we can choose different options that are passed to FRAMA-C. In essence, this acts like an algorithm selection [14] and, e.g., allows us to choose a different configuration of FRAMA-C depending on the specified property.

Harness. Some programs in the SV-Benchmarks collection use specific functions to model non-determinism. We provide implementations for those functions (`__VERIFIER_*`) in a separate C program such that the semantics of those functions can be understood by FRAMA-C. This separate C program is passed to FRAMA-C together with the transformed program p'.

Output Transformer. The output of FRAMA-C needs to be interpreted regarding the original specification, and depending on the outcome, a verification witness needs to be generated. Thus, we need an output transformer for (a) providing a verdict for the verification task and (b) providing a verification witness. Regarding (a), the output transformer interprets the CSV report that can be generated by FRAMA-C to determine whether the program was proven to be safe (verdict `TRUE`), whether a specification violation occurred (verdict `FALSE`), or whether no such statement can be made (verdict `UNKNOWN`). We also generate a minimal correctness or violation witness for the verdicts `TRUE` and `FALSE`, respectively.

The witness automata consist of only one node, which for violation witnesses is marked as violation node. In the future we plan to augment these witnesses with information such as invariants that have been found by Frama-C.

2.2 Frama-C

One of the strengths of Frama-C is its modular architecture [10], which allows a configuration of the best possible analysis backends for a certain verification problem. We choose the plug-in EVA [9], which is well suited for an automatic analysis. Other plug-ins such as the Weakest-Preconditions (WP) plug-in require hints from the user in order to be effective. In the following we will briefly describe the most important aspects of the EVA analysis configuration that we use. For a more detailed description, we refer the reader to the relevant literature [7, 8, 9].

Frama-C provides a meta-option called `-eva-precision` for the EVA plug-in with possible values ranging from 0 to 11. With higher values for this option more precise domains and thresholds are used, at the cost of increased computation time. We currently use the maximum value of 11 in order to make the best use of the 900 s CPU time limit. In the future we might want to iteratively increase this value starting at lower precisions.

Domains. The EVA analysis always uses the domain *cvalue*, which tracks values of variables either as constant values, sets, or intervals of possible values (including modular congruence constraints). For pointer addresses, these are either tracked as addresses with offsets or as so-called garbled mix, which overapproximates the set of possible memory locations. In addition, depending on the precision level, various other domains are used that we describe in the following. The domain *symbolic-locations* tracks a map of symbolic locations to values, which is, e.g., helpful for analyzing expressions containing array accesses such as `a[i]<a[j]`. The *equality* domain tracks equalities of C expressions found in the code, whereas the *gauges* domain tracks relations between variables in a loop with the goal to discover linear inequality invariants [16]. Lastly the *octagon* domain tracks certain linear constraints between pairs of variables [13]. As we use the highest precision level, all of these domains are used in our contribution.

Precision of the State-Space Exploration. Apart from the domains, the precision of state-space exploration in Frama-C is affected by various options. We will describe some of these in the following; a complete list of affected settings and values is always printed by Frama-C when the option *eva-precision* is specified by the user. Option *slevel* (set to 5 000) determines how many separate states are kept before new states will be joined into existing ones. Option *ilevel* (set to 256) determines how many different values are tracked per variable before overapproximating the value range. Option *plevel* (set to 2 000) affects the size up to which arrays are tracked. The option *auto-loop-unroll* (set to 1 024) will determine up to which bound a loop is considered for unrolling.

3 Strengths and Weaknesses

The competition contribution shows the strengths of Frama-C in checking C programs for overflows and also —in the currently supported sub-categories [1]— for reachability. Here we are able to show that our results are comparable and often surpass those of other tools based on abstract interpretation [11] such as Goblint [15]. While the EVA analysis of Frama-C that we use is based on abstract interpretation, the precision options described in Sect. 2.2 allow for a more precise state-space exploration, which behaves more like model checking. More details about the results can be found in the competition report [2] and artifact [3].

The approach that we describe in this paper creates a compatibility layer between the abilities used by Frama-C and the standards used in the SV-Benchmarks collection. While still a work in progress, we have shown that it is possible to bridge this gap while preserving overall soundness. It is also interesting to consider the results on verification tasks from the SV-Benchmarks collections for a tool that did not participate before.

Although our approach is sound in general, we are likely not showcasing the full potential of Frama-C. One aspect to consider here is the large configuration space, which means there might be ways to verify more tasks with a better heuristic for selecting the configuration options. The other aspect is that Frama-C also provides different plug-ins such as the WP plug-in, which requires more (manual) annotations, but can also potentially solve more tasks than the more automatic EVA plug-in.

4 Software Project and Contributors

The software project Frama-C is developed at https://git.frama-c.com/pub/frama-c/ and our adapter Frama-C-SV is developed at https://gitlab.com/sosy-lab/software/frama-c-sv, both being released under open-source licenses. The exact version of the adapter that participated in SV-COMP 2022 is also archived in the competition's tool-archive repository [2] [6]. Frama-C was funded by the European Commission in program Horizon 2020. The adapter Frama-C-SV was funded by the DFG. We thank the Frama-C authors [3] for their contribution to the software-verification community.

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [2] and available on the competition web site. This includes the verification tasks [4], competition results [3], verification witnesses [5], scripts, and instructions for reproduction. The version of Frama-C-SV as used in the competition is archived together with other participating tools [6].

Funding Statement. This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

[1] We opted out of subcategories with unsound results caused by Frama-C making assumptions that are different from the conventions of SV-COMP.

[2] https://gitlab.com/sosy-lab/sv-comp/archives-2022/blob/svcomp22/2022/frama-c-sv.zip

[3] https://frama-c.com/html/authors.html

References

1. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language version 1.17 (2021), available at https://frama-c.com/download/acsl-1.17.pdf
2. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). Springer (2022)
3. Beyer, D.: Results of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831008
4. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831003
5. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5838498
6. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
7. Blazy, S., Bühler, D., Yakobowski, B.: Structuring abstract interpreters through state and value abstractions. In: Proc. VMCAI. pp. 112–130. LNCS 10145, Springer (2017). https://doi.org/10.1007/978-3-319-52234-0_7
8. Bühler, D.: Structuring an Abstract Interpreter through Value and State Abstractions: EVA, an Evolved Value Analysis for Frama-C. Ph.D. thesis, University of Rennes 1, France (2017), available at https://tel.archives-ouvertes.fr/tel-01664726
9. Bühler, D., Cuoq, P., Yakobowski, B., Lemerre, M., Maroneze, A., Perelle, V., Prevosto, V.: Eva: The Evolved Value Analysis plug-in (2020), available at https://frama-c.com/download/frama-c-eva-manual.pdf
10. Correnson, L., Cuoq, P., Kirchner, F., Maroneze, A., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C user manual (2020), available at https://frama-c.com/download/frama-c-user-manual.pdf
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL. pp. 238–252. ACM (1977)
12. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
13. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1
14. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976). https://doi.org/10.1016/S0065-2458(08)60520-3
15. 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: Proc. TACAS (2). pp. 438–442. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_28
16. Venet, A.: The gauge domain: Scalable analysis of linear inequality invariants. In: Proc. CAV. pp. 139–154. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_15