Partial Verification and Intermediate Results as a Solution to Combine Automatic and Interactive Verification Techniques

Dirk Beyer

LMU Munich, Munich, Germany

Abstract. Many of the current verification approaches can be classified into automatic and interactive techniques, each having different strengths and weaknesses. Thus, one of the current open problems is to design solutions to combine the two approaches and accelerate technology transfer. We outline four existing techniques that might be able to contribute to combination solutions: (1) Conditional model checking is a technique that gives detailed information (in form of a condition) about the verified state space, i.e., informs the user (or tools later in a tool chain) of the outcome. Also, it accepts as input detailed information (again as condition) about what the conditional model checker has to do. (2) Correctness witnesses, stored in a machine-readable exchange format, contain (partial) invariants that can be used to prove the correctness of a system. For example, tools that usually expect invariants from the user can read the invariants from such correctness witnesses and ask the user only for the remaining invariants. (3) Abstraction-refinement based approaches that use a dynamically adjustable precision (such as in lazy CEGAR approaches) can be provided with invariants from the user or from other tools, e.g., from deductive methods. This way, the approach can succeed in constructing a proof even if it was not able to come up with the required invariant. (4) The technique of path invariants extracts (in a CEGAR method) a path program that represents an interesting part of the program for which an invariant is needed. Such a path program can be given to an expensive (or interactive) method for computing invariants that can then be fed back to a CEGAR method to continue verifying the large program. While the existing techniques originate from software verification, we believe that the new combination ideas are useful for verifying general systems.

1 Introduction

Automatic verification techniques usually expect the user to set parameters, and the prover computes the necessary invariants and the proof — the strength of this technique is that it works for large systems. Interactive verification techniques usually expect the user to provide invariants and the prover establishes a formal correctness proof — the strength of this technology is that it works for sophisticated specifications. In order to increase the impact of formal verification, we need approaches that combine the advantages of both. The organizers

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part I, LNCS 9952, pp. 874–880, 2016.
DOI: 10.1007/978-3-319-47166-2_60

of the ISoLA 2016 track on "Correctness-by-Construction and Post-hoc Verification" [36] emphasize the importance of bringing together researchers from different verification communities, in order to exchange ideas and discuss ways to combine techniques and improve the overall verification process. Bringing together different communities and develop verification tools that integrate solutions from various viewpoints takes time and requires a long series of such meetings (for example, a similar event with the same objective took place at Dagstuhl a few years ago [15]). A similar "joining effort" of two communities took place in the past: the research areas of data-flow analysis and software model checking were originally using separate concepts, techniques, and algorithms, but were unified in the past two decades [9,33]. Today, tools for automatic software verification usually combine techniques from data-flow analysis with techniques from software model checking (e.g., [11,16,19,22–24,29,34,35,38]).

Maturity Level of Research Areas. Both automatic and interactive verification are mature research areas. This is not only witnessed by the many valuable publications (cf. surveys [4,28] for an overview), but in particular by the large set of available tools that make it possible to actually verify real software with the help of new technology [2,30]. There are several well-maintained software projects that reflect the state of the art in the area of automatic verification, for example, Blast [11], Cbmc [19], CPAchecker [16], Slam [3], and Ultimate [24]; a large list of recent tool implementations can be found in the SV-COMP competition report [5]. Also in interactive verification, the state of the art is available in well-maintained software projects, for example, AutoProof [37], Dafny [31], KeY [1], Kiv [20], and VeriFast [27]; a larger list can be found in the VerifyThis competition report [26].

There are four international competitions in the area of software verification, which all have the goal to showcase the strengths and abilities of the latest technology, and at the same time identify the limits of the existing approaches. Rers [25] is a competition on verification of generated event-condition-action programs. This allows to control the features that are used in the program and for which support is needed during the verification process. The goal is to identify the overall current abilities of software verifiers, without any restriction of the process or of the resources. SV-COMP [5] is a controlled experiment to measure effectiveness and efficiency of fully-automatic software verification. Verifiers are executed without interaction on a dedicated computing environment and with limited, controlled computing resources (CPU time, memory). termComp [21] focuses on the particular specification of termination. VerifyThis [26] concentrates on evaluating different verification approaches and ideas to formalize a given problem, i.e., develop a model and a specification and then prove correctness.

Outline. This article presents a position statement that was prepared for the ISoLA 2016 meeting. We use a few existing approaches from the viewpoint of automatic verification which might be able to contribute to combination approaches. We outline four solutions to combine automatic verification with interactive verification by exchanging partial and intermediate verification results using well-defined interfaces.

2 Exchanging Partial and Intermediate Results

Conditional Model Checking. In classical model checking, the outcome of a model checker is either TRUE or FALSE. In practice, however, executions of classical model checkers often end without delivering any useful result (tool gives up, component or tool crashes, tool runs out of resources), which means that the resources that the user spent on the verification task are lost without any benefit for the user. *Conditional model checking* [12] is a technique that gives detailed information (in form of a condition) about the verified state space, i.e., informs the user or tools later in the tool chain of the outcome. Also, it accepts as input detailed information (again as condition) about what the conditional model checker has to do, i.e., which parts of the state space to verify.

The idea to use a sequential combination of different approaches is not new, for example, a combination of CCURED [32] with BLAST [11] was explored more than ten years ago: in a first phase, CCURED added run-time checks to the program in order to make sure that no memory-safety violation happens without run-time notification of the user; in a second phase, BLAST removed all run-time checks that it was able to verify statically [10]. All run-time checks that could not be verified remained in the reduced "cured" program. Conditional model checking formalized the approach and emphasizes the flow of information about what is still to be verified between different checkers.

If developers of automatic and interactive verification tools agree on an exchange format for conditions that describe the state-space that is to be verified, then many verifiers (both automatic and interactive) can be turned into conditional model checkers.

Correctness Witnesses. Until recently, model checkers reported counterexample traces in proprietary formats, mostly in formats that were difficult to read, not only for users but also for machines, i.e., the reported counterexamples were sometimes difficult to inspect and thus of limited use. *Error witnesses* [8], stored in an exchangeable standard format, overcome this problem. Witnesses can now be inspected by users and tools without knowledge about the implementation of the verifier that produced the witness, and the trust in the verification result can be increased by independent witness validators. The witnesses can also be visualized and used for debugging [6]. Further extending this concept, *correctness witnesses* [7] store hints for establishing a correctness proof. Program invariants (perhaps partial invariants) are stored in a machine-readable exchange format.

In a combination scenario, tools that usually expect invariants from the user can be provided with invariants from correctness witnesses. This way, automatic tools can compute as many invariants as possible automatically, the resulting invariants are provided to the interactive tool as input, and the interactive tool needs to ask only for the remaining invariants. While interactive tools already compute some invariants automatically, the exchange with automatic verifiers accelerates technology transfer and adoption of implementations with less effort.

Precisions. One of the key challenges in automatic software verification is to algorithmically compute an abstract model that is precise enough to be able to prove that the specification holds and that is at the same time coarse enough to make the verification process efficient. The level of abstraction of the abstract model can be expressed as a *precision* [14]. The precision is often computed using counterexample-guided abstraction refinement (CEGAR) [18]. An *infeasible error path* is an error path through the to-be-verified program that is possible in the abstract model, but not in the concrete program, i.e., the precision of the abstract model is too coarse. CEGAR uses infeasible error paths to derive useful information for refining the abstract model, i.e., to increase the precision. For many abstract domains that are used with dynamic abstraction refinement (predicates, variable assignments, shape graphs, intervals), the precision can be stored for later reuse [17], for example, for regression verification.

In case of an incomplete verification run, an automatic verifier was perhaps identifying the correct variables that the invariant should talk about, but the constructed precision was not correctly establishing the relation of the variables. For example, consider the code snippet in Fig. 1 (meant as a part of a very large program) and assume that CEGAR with predicate abstraction brought up an infeasible error path that goes once through the body of the loop and then violates the assertion (due to not yet tracking any variables). For a human it might be easy to see that the invariant $x = y$ is needed at the loop head in order to prove the correctness of the program, while the value of the unknown constant n is irrelevant for the safety property and can be abstracted away. But an interpolation-based refinement procedure might unluckily come up with interpolants that contain predicates like $x = 0$, $y = 0$, $x = 1$, $y = 1$, which are sufficient to eliminate the current infeasible error path, but in the next CEGAR iteration, an infeasible error path that goes twice through the loop body will be brought up, and so on. This (rather simple) automatic approach would fail because it is not able to generalize the information from the path to the loop invariant $x = y$.

```
1  x = 0;
2  y = 0;
3  while (x < n) {
4      x++;
5      y++;
6  }
7  assert(x==y);
```

Fig. 1. Code snippet that requires loop invariant $x = y$

So it would be an interesting approach to interactively tell the user (or a different tool) that an invariant is needed that talks about variables x and y. Then, either an interactive prover is fed with the invariants that were computed by the automatic verifier together with the additional invariants from the user, or the automatic verifier is restarted with the additional invariants. Together, the different approaches might be able to completely solve the verification task.

Path Invariants. Sometimes, adding a certain information about the path to the precision is sufficient to eliminate the infeasible error path from further exploration, but not other error paths that are infeasible for a similar reason (cf. explanation of the example of Fig. 1 above). The approach of *path invariants* [13] constructs a *path program* (hopefully much smaller than the original program)

that contains the infeasible error path, and in addition many similar error paths. Now, such a path program for which an invariant is needed can be given to an expensive method for computing invariants, and the invariants can then be fed back to a CEGAR method to refine the precision and continue verifying the large original program. The loop invariants for the path program will eliminate a whole series of infeasible error paths, instead of only one single infeasible error path.

If an automatic verifier is not able to derive an invariant and would have to abort the verification process, it could instead ask the user for an appropriate invariant. Since the path program is small and focuses on the reason for which the automatic verifier was not able to construct an invariant, a user can perhaps use an interactive verifier to construct an invariant for the path program and feed this back to the automatic verifier. The advantage over the precision-based solution above is that the user (or interactive tool) is given the isolated, but full context of a complete (path) program.

3 Conclusion

Currently, automatic techniques can verify large systems, but with rather simple specifications, while interactive techniques can verify complicated specifications, but only for systems of rather limited size. To further improve the verification technology, we need solutions to combine the techniques from automatic and interactive verification. We have outlined a few new ideas for combining very different verification approaches using existing techniques that support partial verification and the exchange of intermediate verification results. To further stimulate the discussion and develop new combination ideas, it is necessary to implement the above-mentioned combination ideas and report experimental results.

References

1. Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P.H., Ulbrich, M.: The KeY platform for verification and analysis of Java programs. In: Giannakopoulou, D., Kröning, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 55–71. Springer, Heidelberg (2014)
2. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with Slam. Commun. ACM **54**(7), 68–76 (2011)
3. Ball, T., Rajamani, S.K.: The Slam project: Debugging system software via static analysis. In: POPL 2002, pp. 1–3. ACM (2002)
4. Beckert, B., Hähnle, R.: Reasoning and verification: State of the art and current trends. IEEE Intell. Syst. **29**(1), 20–29 (2014)
5. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 887–904. Springer, Heidelberg (2016)
6. Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 502–509. Springer, Heidelberg (2016)

7. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: FSE 2016. ACM (2016)
8. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: FSE 2015, pp. 721–733. ACM (2015)
9. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Clarke, E.M., Henzinger, T.A., Veith, H. (eds.) Handbook on Model Checking. Springer (to appear, 2017)
10. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: Checking memory safety with Blast. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 2–18. Springer, Heidelberg (2005)
11. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer **9**(5–6), 505–525 (2007)
12. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: FSE 2012. ACM (2012)
13. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI 2007, pp. 300–309. ACM (2007)
14. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: ASE 2008, pp. 29–38. IEEE (2008)
15. Beyer, D., Huisman, M., Klebanov, V., Monahan, R.: Evaluating software verification systems: Benchmarks and competitions (Dagstuhl reports 14171). Dagstuhl Rep. **4**(4), 1–19 (2014)
16. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
17. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: ESEC/FSE 2013, pp. 389–399. ACM (2013)
18. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
19. Clarke, E., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
20. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: Kiv: Overview and VerifyThis competition. Int. J. Softw. Tools Technol. Transfer **17**(6), 677–694 (2015)
21. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition. In: Felty, A.P., Middeldorp, A. (eds.) CADE-25. LNCS, vol. 9195, pp. 105–108. Springer, Heidelberg (2015)
22. Albarghouthi, A., Gurfinkel, A., Li, Y., Chaki, S., Chechik, M.: Ufo: Verification with interpolants and abstract interpretation (Competition Contribution). In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 637–640. Springer, Heidelberg (2013)
23. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: A framework for verifying C programs (Competition Contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 447–450. Springer, Heidelberg (2015)
24. Heizmann, M., Dietsch, D., Greitschus, M., Leike, J., Musa, B., Schätzle, C., Podelski, A.: Ultimate automizer with two-track proofs (Competition Contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 950–953. Springer, Heidelberg (2016)

25. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D., Păsăreanu, C.S.: Rigorous examination of reactive systems. The RERS challenges 2012 and 2013. Int. J. Softw. Tools Technol. Transfer **16**(5), 457–464 (2014)
26. Huisman, M., Klebanov, V., Monahan, R., Tautschnig, M.: VerifyThis 2015: A program verification competition. Int. J. Softw. Tools Technol. Transfer (2016)
27. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
28. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), Article No. 21 (2009)
29. Karpenkov, E.G.: Lpi: Software verification with local policy iteration (Competition Contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 930–933. Springer, Heidelberg (2016)
30. Khoroshilov, A., Mutilin, V., Petrenko, A., Zakharov, V.: Establishing Linux driver verification process. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 165–176. Springer, Heidelberg (2010)
31. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
32. Necula, G.C., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy code. In: POPL 2002, pp. 128–139. ACM (2002)
33. Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998)
34. Schrammel, P., Kröning, D.: 2ls for program analysis (Competition Contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 905–907. Springer, Heidelberg (2016)
35. Ströder, T., Aschermann, C., Frohn, F., Hensel, J., Giesl, J.: AProVE: Termination and memory safety of C programs (Competition Contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 417–419. Springer, Heidelberg (2015)
36. ter Beek, M., Hähnle, R., Schaefer, I.: Correctness-by-construction and post-hoc verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016 Part I. LNCS, vol. 9952, pp. 723–729. Springer, Heidelberg (2016)
37. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: Auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015)
38. Zheng, M., Edenhofner, J.G., Luo, Z., Gerrard, M.J., Rogers, M.S., Dwyer, M.B., Siegel, S.F.: CIVL: Applying a general concurrency verification framework to C/Pthreads programs (Competition Contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 908–911. Springer, Heidelberg (2016)