2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00215

CEGAR-PT: A Tool for Abstraction by Program Transformation

Dirk Beyer
LMU Munich, Germany

Marian Lingsch-Rosenfeld
LMU Munich, Germany

Martin Spiessl
LMU Munich, Germany

***Abstract*—Abstraction is an important approach for proving the correctness of computer programs. There are many implementations of this approach available, but unfortunately, the various implementations are difficult to reuse and combine, and the successful techniques have to be re-implemented in new tools again and again. We address this problem by contributing the tool CEGAR-PT, which views abstraction as program transformation and integrates different verification components off-the-shelf. The idea is to use existing components without having to change their implementation, while still adjusting the precision of the abstraction using the successful CEGAR approach. The approach of CEGAR-PT is largely general: It only restricts the abstraction to transform, given a precision that defines the level of abstraction, one program into another program. The abstraction by program transformation can over-approximate the data flow (e.g., havoc some variables, use more abstract types) or the control flow (e.g., loop abstraction, slicing). To illustrate our tool, we provide a demonstration video, accessible at https://youtu.be/ASZ6hoq8asE.**

I. Introduction

Program transformations are a promising approach for improving the capabilities of software verifiers [1, 2, 3, 4, 5, 6]. These program transformations are applied on the source code of the program, producing a new, transformed program, which over-approximates the original behavior (or maintains it equivalent) with the goal of making the verification easier. Figure 1 shows an example of such a program transformation.

```
int main() {
   int y = nondet();
   if (y < 100) {
      while (y < 10000) {
         y += 1;
      }
      if (y != 10000) {
         error();
      }
   }
}
```

(a) Original program

```
int main() {
   int y = nondet();
   if (y < 100) {

      y = 10000;

      if (y != 10000) {
         error();
      }
   }
}
```

(b) Simpler but equivalent program

Fig. 1: Loop abstraction by source-code transformation

Due to the variety and flexibility of program transformations, we need to address multiple concerns when using them: How to make use of the program transformations as part of the verification process? How to ensure the modified program correctly models the original program's semantics? How to choose the most suitable program transformations?

Program transformations are independent from the verifier being used. This makes it possible to use the transformed programs as input for several verifiers, increasing reusability of the program transformation. This is especially important since such transformations are usually difficult to implement and bugs could result in unsoundness of the verifier.

In order to ensure that the answer of a verifier for a transformed program also applies to the original program, we need to ensure that the transformation is over-approximating. The correctness of a specific transformations [2, 5] can be ensured by formal proofs. As a complementing technique, we can also leverage verification witnesses [7, 8] and check whether the result can be validated on the original program with the information that was computed as witness for the transformed program.

While program transformations are widely used [1, 2, 3, 4], the way how these can be combined is still an open question. A possible solution for this problem is to use counterexample-guided abstraction refinement (CEGAR) [9, 10] to select a suitable level of abstraction. This allows us to automatically select program transformations that are useful for the verification task at hand, while ruling out transformations that are on the wrong abstraction level.

In order to address these concerns, we present a modular CEGAR approach for applying program transformations for verification, and provide an implementation in our tool CEGAR-PT. We decompose the verification approach into four different components, which allows for reuse of existing off-the-shelf components, as inspired by the unifying component framework for cooperative verification [11].

Contribution. We provide the following contributions:

- CEGAR-PT can integrate any *separately implemented* program transformation in CEGAR.
- CEGAR-PT enables CEGAR for *automatic selection* of suitable program transformations.

Related Work. CEGAR-PT adopts the approach of counterexample-guided abstraction refinement (CEGAR) [9, 10] to loop abstraction in a modular way. There are two very related results, which paved the road for our investigation: C-CEGAR [12] implements CEGAR in a modular way, using off-the-shelf components. The precision in this approach consists of precisions for predicate-based abstractions, such as predicate abstraction and trace abstraction. The main insights are that the overhead of modularization is not prohibitive, and

DOI 10.1109/ASE56229.2023.00215

that the effectiveness can be increased by leveraging a more diverse set of verification components.

Unified Loop Abstraction [2] proposes to use CEGAR for loop abstractions inside a specific verifier. The main insight is that it is possible to use control-flow refinement using CEGAR. The approach unifies several approaches for loop abstraction in one framework as control-flow abstraction. CEGAR-PT goes a step further and uses off-the-shelf program transformations in the refinement step of CEGAR. In other words, CEGAR-PT is similar to C-CEGAR, but the abstractions are not data-flow abstractions but control-flow abstractions.

Symbolic Computation via Program Transformations [6] explores a way to apply program transformations separately from state-space exploration. While we also aim at separating these concerns, we employ CEGAR to choose between various applicable program transformations while they only discuss CEGAR inside the state-space exploration.

II. Background

Program Transformations. Program transformations find their use in a wide variety of application areas [13, 14]. For aiding in program verification, we distinguish between local and global transformations. Global transformations essentially change the whole program and aid a specific analysis, e.g., symbolic computation [4], sequentialization of concurrency [15], or removing complicated language constructs [16]. Local transformations aim at improving the verifiability of certain local program statements such as loops.

Here we concentrate on local transformations of program loops that are already available in CPAchecker [2], namely "Constant Extrapolation", "Havoc Abstraction", "Naive Abstraction", and "Output Loop Abstraction".

Verification Witnesses. Witnesses are artifacts that can be produced by a verifier together with its verdict. A well-established format [7, 8] defines witnesses as protocol automata relative to the program's control-flow automata (CFA) that either track paths to the discovered specification violation (called violation witnesses) or invariants that are useful for proving the program correct (called correctness witnesses).

CEGAR. Counterexample-guided abstraction refinement (CEGAR) [9, 10] is an algorithm that starts with an abstraction over-approximating the behavior of a program. Using this abstraction it attempts to prove or disprove that program P satisfies specification φ. Once a potential counterexample has been found, its feasibility is checked. If the counterexample is feasible, an alarm is raised and CEGAR terminates. If the counterexample is infeasible, then the abstraction is refined using information learned from the counterexample and the process starts again. The abstraction and its refinements are defined by the precision. Figure 2 illustrates this cyclic process.

III. CEGAR for Program Transformations

While state-of-the-art verification tools usually track the abstraction as data-flow domains (e.g., predicates [17]) over the program states, control-flow abstractions can also be encoded as abstractions used in CEGAR [2]. CEGAR-PT integrates external, off-the-shelf control-flow transformations into CEGAR, i.e., we eliminate the dependency on the internal program representation of a particular verifier. This allows any off-the-shelf verifier to make use of the control-flow abstractions.

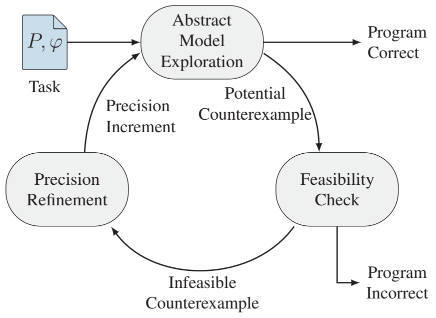


Fig. 2: General workflow of CEGAR (taken from [12])

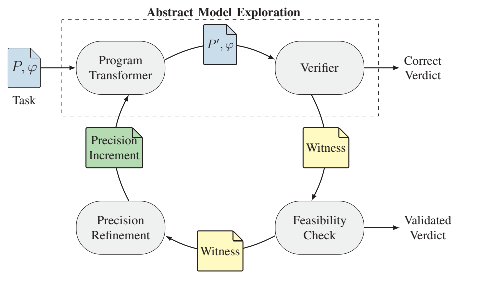
In order to make program transformations available to all verifiers, we decomposed the CEGAR loop into multiple components, similar to [12]. These components interact via well-defined interfaces, which makes it easy to replace and compare different implementations of the same component. Figure 3a shows this decomposition. For simplicity, the precision π, which is passed through every step in the loop, and modified by the program transformer, is omitted in Fig. 3a.

Each component will be introduced in a general context, followed by a description on how they are instantiated in the tool demonstration.

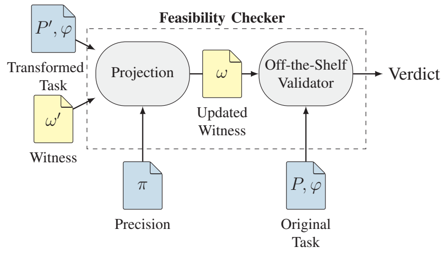
1) Program Transformer: This component takes as input a program and a precision increment, and produces as output a new, transformed program and a precision. The precision can be used to guide the verifier or validator. Using the precision increment, the program transformer can determine the next control-flow abstraction to be applied in order to improve the level of abstraction.

CEGAR-PT uses patches produced by CPAchecker's loop abstraction [2] to implement the program transformer. The transformations are applied to the original program P in order to produce a new program P'. Which transformation (which patch) to use is determined by the precision π and the precision increment. The transformation either over-approximates the concrete program semantics, in which case it is called an over-approximating transformation, or leaves it equivalent, in which case it is called a precise transformation.

2) Verifier: Calls to a software verifier are encapsulated using CoVeriTeam [18], which provides a common interface for using off-the-shelf verifiers. It allows the usage of at least all 45 verifiers for C that participated in the competition on software verification [19]. The verdict that the verifier produces corresponds to the transformed program P'. In order to translate this verdict to a verdict for the original program P, we use the information about P' that the program transformer included in the precision. This information tells us which verdicts can be returned and which need to be validated.



(a) Workflow of CEGAR for program transformations



(b) Components of a feasibility checker

Fig. 3: Structure of the various components of CEGAR-PT

Since all used transformations of CEGAR-PT are either over-approximating or precise, we can directly report proofs. Alarms need to be validated, as explained in the following section.

3) Feasibility Check via Witnesses: A counterexample for a transformed program P' whose state space over-approximates the state space of the original program P cannot be returned directly to the user, because the counterexample for P' could be infeasible for P.

In order to solve this problem, our feasibility checker converts the witness from the verifier of the transformed program P' to a witness for the original program P and validates if the verdict provided by the verifier applies to P as well. Therefore, the feasibility checker is composed of two components: (1) a projection operator that uses the witness of P', specification φ, and the precision π to produce a witness w for P and φ, and (2) a component from any off-the-shelf witness validator (there are at least ten available [19]) that validates the witness w.

The precision tracks which transformations were applied. This allows us to determine which program lines present in the witness were not from the original program. Removing these lines allows us to get a witness for the original program, i.e., project the witness for pair P', φ to a witness for P, φ.

4) Precision Refinement: In order to refine the choice of program transformations being used, information can be extracted from the produced witness. This allows the program transformer to learn from previously failed transformations how to make the abstraction (and the transformed program) more precise.

Our approach can use multiple (loop-abstraction) transformations. These transformations are produced and applied to the original program by the program transformer. The precision tracks which transformations were applied, while the precision increment tracks which transformations were ruled out and which ones can still be applied. Since we only consider transformations that over-approximate or keep the program semantics equivalent, we only need to deal with violation witnesses from the verifier. In order to determine the precision increment, the precision refinement selects one transformation whose modified program lines coincide with the program lines present on the error path of the violation witness. It then selects the next transformation to be used for that part of the program. The identity function, resulting in the original program, is considered as last transformation.

5) Precision: The precision contains information about the process of producing a new, transformed program P'. This information can be used to see which verdicts of the verifier can be returned as is and which need to be validated. It can also be used to inform the validator how to transform the witness of the transformed program into a witness of the original program.

The precision introduces a dependency between the feasibility checker and the program transformer, because the feasibility checker needs to project the witness into one for the original program. This still allows for use of an arbitrary witness validator, but the projection must consider what program transformer is being used.

In our demonstration, which uses CPACHECKER as a program transformer, the precision remembers which transformations had been applied to the original program, what type of transformations they were (i.e., over-approximating or precise), and which program lines of the modified program were part of the original program. This allows all the other components to determine the information they need.

IV. TOOL DEMONSTRATION

In order to illustrate CEGAR-PT we provide a demonstration in form of a screencast[1]. The screencast shows an example run of CEGAR-PT, using CPACHECKER as the off-the-shelf verifier, to verify a program with multiple loops. Each loop is replaced by one of the loop abstractions made available by CPACHECKER. Afterwards the intermediate steps of CEGAR-PT during this execution are shown by using the debug option. Finally, the example is also verified using SYMBIOTIC as a verifier instead of CPACHECKER.

V. EVALUATION

A full evaluation of CEGAR-PT and comparison using different verifier backends can be found on our supplementary webpage[2]. This supplementary evaluation aims to answer the following two research questions:

- **RQ 1:** Do the program transformations improve the number of tasks that can be solved by off-the-shelf verifiers at a negligible overhead?
- **RQ 2:** How do the execution times compare between using program transformations in a modular setting to using them inside the verifier?

[1]https://youtu.be/ASZ6hoq8asE
[2]https://www.sosy-lab.org/research/cegar-pt/

VI. Applications

We now explain application scenarios for CEGAR-PT.

Reuse of existing program-transformation techniques. Program transformations that are already implemented in one tool can be made available to other tools by turning them into a separate, externally usable component. This allows transformations to be quickly used by other tools without having to reimplement them for other internal representations. Developers of verifiers can this way quickly evaluate if such a program transformation is beneficial for their tool and if it is, they may decide to implement it inside their tool.

Development of new program transformations. Developing program transformations can be difficult. Using CEGAR-PT, this process can be improved, by first implementing the abstraction as program transformation in CEGAR-PT, evaluate this implementation, and if this improves the verification process, it can then be used as program transformation, or implemented in the verifier of choice. Especially considering that CEGAR-PT is written in Python, it is easy to quickly prototype different approaches using it.

Development of selection heuristics. In order to select the best program transformation to be used when a refinement occurs in the CEGAR loop, a selection heuristic is necessary. Due to the modular nature of CEGAR-PT, more program transformations can be used in order to determine a more general or better selection heuristic.

VII. Conclusion

Program transformations have been shown to increase the number of problems that a verifier can solve (see, e.g., [20]). In order to make program transformations reusable, and to simplify their development process, we developed the tool CEGAR-PT. This new tool integrates external program transformations into the CEGAR loop, which can be used with any off-the-shelf program transformation and any off-the-shelf verifier. The CEGAR loop also checks the feasibility of the counterexamples.

Using a modular approach introduces a small performance cost for simple tasks, compared to directly manipulating the control-flow structure inside a verifier. Nonetheless, it allows for the reuse and simpler implementation of program transformations, which is much more important than the relatively small performance gain.

It has been a research question since long to identify intermediate results that can be made available for monitoring and understanding of the verification process. We believe that using the programming language of the input program as exchange format is a choice that easily enables exchange.

Data-Availability Statement. The tool CEGAR-PT is licensed under the open-source license Apache 2.0. The tool is available at https://gitlab.com/sosy-lab/software/controlflow-cegar, and we prepared a reproduction package [21] to explore the features of our tool and to replay the demonstration. The description of the evaluation experiments and the corresponding data are available on our supplementary web page.[2]

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

References

[1] Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141. IEEE (2019). doi:10.1109/ASE.2019.00121

[2] Beyer, D., Rosenfeld, M.L., Spiessl, M.: A unifying approach for control-flow-based loop abstraction. In: Proc. SEFM. pp. 3–19. LNCS 13550, Springer (2022). doi:10.1007/978-3-031-17108-6_1

[3] Alglave, J., Kröning, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Proc. ESOP. pp. 512–532. LNCS 7792, Springer (2013). doi:10.1007/978-3-642-37036-6_28

[4] Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. pp. 313–332. LNCS 11187, Springer (2018). doi:10.1007/978-3-030-02508-3_17

[5] Steinhöfel, D.: REFINITY to model and prove program transformation rules. In: Proc. APLAS. pp. 311–319. LNCS 12470, Springer (2020). doi:10.1007/978-3-030-64437-6_16

[6] Lauko, H., Rockai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. pp. 313–332. LNCS 11187, Springer (2018). doi:10.1007/978-3-030-02508-3_17

[7] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). doi:10.1145/2786805.2786867

[8] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). doi:10.1145/2950290.2950351

[9] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV. pp. 154–169. LNCS 1855, Springer (2000). doi:10.1007/10722167_15

[10] Ball, T., Rajamani, S.K.: Boolean programs: A model and process for software analysis. Tech. Rep. MSR Tech. Rep. 2000-14, Microsoft Research (2000), https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2000-14.pdf

[11] Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). doi:10.1007/978-3-030-61362-4_8

[12] Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In: Proc. ICSE. pp. 536–548. ACM (2022). doi:10.1145/3510003.3510064

[13] Partsch, H., Steinbrüggen, R.: Program transformation systems. ACM Comput. Surv. **15**(3), 199–236 (1983). doi:10.1145/356914.356917

[14] Visser, E.: A survey of strategies in program transformation systems. In: Proc. WRS. pp. 109–143. ENTCS 57, Elsevier (2001). doi:10.1016/S1571-0661(04)00270-1

[15] Fischer, B., Inverso, O., Parlato, G.: CSeq: A concurrency pre-processor for sequential C verification tools. In: Proc. ASE. pp. 710–713. IEEE (2013). doi:10.1109/ASE.2013.6693139

[16] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of C programs. In: Proc. CC. pp. 213–228. LNCS 2304, Springer (2002). doi:10.1007/3-540-45937-5_16

[17] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL. pp. 58–70. ACM (2002). doi:10.1145/503272.503279

[18] Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). doi:10.1007/978-3-030-99524-9_31

[19] Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). doi:10.1007/978-3-031-30820-8_29

[20] Slabý, J., Strejček, J., Trtík, M.: Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In: Proc. FMICS. pp. 207–221. LNCS 7437, Springer (2012). doi:10.1007/978-3-642-32469-7_14

[21] Beyer, D., Rosenfeld, M.L.: Reproduction package for ASE 2023 article 'CEGAR-PT: A tool for abstraction by program transformation'. Zenodo (2023). doi:10.5281/zenodo.8287183