# A Unifying Approach for Control-Flow-Based Loop Abstraction

Dirk Beyer, Marian Lingsch Rosenfeld, and Martin Spiessl

LMU Munich, Germany

**Abstract.** Loop abstraction is a central technique for program analysis, because loops can cause large state-space representations if they are unfolded. In many cases, simple tricks can accelerate the program analysis significantly. There are several successful techniques for loop abstraction, but they are hard-wired into different tools and therefore difficult to compare and experiment with. We present a framework that allows us to implement different loop abstractions in one common environment, where each technique can be freely switched on and off on-the-fly during the analysis. We treat loops as part of the abstract model of the program, and use counterexample-guided abstraction refinement to increase the precision of the analysis by dynamically activating particular techniques for loop abstraction. The framework is independent from the underlying abstract domain of the program analysis, and can therefore be used for several different program analyses. Furthermore, our framework offers a sound transformation of the input program to a modified, more abstract output program, which is unsafe if the input program is unsafe. This allows loop abstraction to be used by other verifiers and our improvements are not 'locked in' to our verifier. We implemented several existing approaches and evaluate their effects on the program analysis.

**Keywords:** Software verification · Program analysis · Loop abstraction · Precision adjustment · Counterexample-guided abstraction refinement · CPAchecker

## 1 Introduction

Software programs are among the most complex systems that mankind produces. Programs tend to have a complex state space and hence verifying the correctness of software programs is a difficult task. Abstraction is a key ingredient to every successful approach to prove the correctness of large programs. Let us look at a few examples: Constant propagation [21] abstracts from concrete values for a variable if the value of the variable is not constant. Counterexample-guided abstraction refinement (CEGAR) [14] is an algorithm to incrementally refine the level of abstraction until the abstract model is detailed enough to prove the correctness, while the abstract model is still coarse enough to make the analysis feasible. Predicate abstraction [18, 20] uses an abstract domain where the abstract state is described as a combination of predicates from a certain given precision [8] (a set of predicates). The precision is refined with CEGAR by adding new predicates to the precision. Shape analysis [25] abstracts from concrete data structures on the heap and stores only their shape for the analysis.

Finally, *loop abstraction* is a technique to abstract the behavior of a program with a loop in such a way that the correctness of the abstract program implies the correctness of the original program. There are several approaches for loop abstraction proposed in the literature [15, 16, 19, 22]. While we will concentrate on reachability here, this technique can also be applied to other properties.

We contribute a formalism that treats loop abstraction as an abstraction in the sense of CEGAR: The precision is a choice of a certain approach to loop abstraction (level of abstraction of the loop). If the abstract model of the program defined by this precision (= loop abstraction) is too coarse to prove correctness, then we refine the abstract model by setting the precision to a different (more precise) loop abstraction.

**Example.** Let us consider the small program in Fig. 1a. The program uses one variable `x`, which is initialized with some large, even value and decreased by 2 in a loop. The specification requires that the value of `x` is even after the loop terminates. It is easy for a human to see that an even number, decreased by an even number, always yields an even number, no matter how often this is done. In other words, we discover the invariant that `x` is even and check if it is preserved. However, in this example there exists an even simpler invariant: The data type of `x` is unsigned int, which means values greater or equal to zero. The control flow cannot leave the loop as long as `x` is greater than 0. Once the control flow leaves the loop, we know that the value is 0, and thus, even. The loop-exit condition, together with the above argument, implies the specification. A program analysis that cannot discover this (e.g., bounded model checking, explicit-value analysis, interval analysis) has to unroll the loop many times.

But we can construct the loop abstraction in Fig. 1b, which executes the new body only if the loop condition `x > 0` is fulfilled, and the new body models all behaviors that occur when the original program enters the loop. The new body havocs (sets to an arbitrary value) the variable `x`. Then it constrains the values of `x` by blocking the further control flow if the loop condition still holds, i.e., the original program would stay in the loop. Surprisingly, since the loop-exit condition now implies the specification, this overapproximation of the original program still satisfies the specification.

**Contributions.** This paper makes the following contributions:

- We propose a framework that can express several existing approaches for loop abstraction and makes it possible to compare those different approaches.
- The framework allows to switch dynamically, on-the-fly, between different loop-abstraction techniques, selecting different abstraction levels.
- The framework is independent from the underlying abstract domain of the program analysis. The loop abstractions work using transformations of the control flow. Once implemented, a strategy for loop abstraction is applicable to several abstract domains.

```c
unsigned int x = 0x0ffffff0;
while (x > 0) {
  x -= 2;
}
assert(!(x % 2)));
```

**(a)** Original program

```c
unsigned int x = 0x0ffffff0;
if (x > 0) {
  x = nondet_uint();
  if (x > 0) {
    return 0;
  }
}
assert(!(x % 2)));
```

**(b)** Havoc abstraction

```c
unsigned int x = 0x0ffffff0;
if (x > 0) {
  long long iterations = x/2;
  x -= 2*iterations;
  if (x > 0) {
    x -= 2;
  }
}
assert(!(x % 2)));
```

**(c)** Constant extrapolation

```c
unsigned int x = 0x0ffffff0;
if (x > 0) {
  x = nondet_uint();
  if (x <= 0) {
    return 0;
  }
  x -=2;
  if (x > 0) {
    return 0;
  }
}
assert(!(x % 2)));
```

**(d)** Naive abstraction

Fig. 1: Application of various loop abstraction strategies on the benchmark program `simple_4-2.c` from the SV-Benchmarks set; only the body of the main function is shown here

- We export the modified C program, such that the loop-abstraction techniques can be used by other verifiers.
- The framework is publicly available as an extension of the open-source verification framework CPAchecker.
- We evaluate the effectiveness and efficiency of the framework on a benchmark set from the publicly available collection of verification tasks SV-Benchmarks, and compare it with state-of-the-art tools.

**Related Work.** In the following we discuss the most related existing approaches.

*Loop Acceleration.* As this is an obvious way to speed up verification, many different approaches have been proposed to calculate the effects of a loop execution [17, 19, 26]. We present only a very basic form where we accelerate variables that are incremented by a fixed value in loops with a known number of iterations, since our interest is rather into gaining insights into how different existing approaches can be combined to further improve their usefulness. As such we are interested in implementing other approaches for loop acceleration as strategies into our framework, rather than coming up with new ways of accelerating single loops.

*Loop Abstraction.* While loop acceleration is useful also in other areas, e.g., for compiler optimizations, verifiers have the possibility of using loop abstractions (i.e., overapproximatons) instead, for aiding the generation of correctness proofs. Since loop abstraction is closely related to invariant generation, and this is the main challenge in software verification, there is a large body of literature. We will therefore look at only those publications that also make use of the idea to encode the abstractions into the source code. The abstraction techniques we describe in this paper are taken taken from existing publications [15, 16]. As with loop accelerations, our goal is not to invent new strategies, but rather investigate how existing strategies can be combined. Also VeriAbs [1] uses a variety of loop-abstraction techniques, but only statically generates a program that is then checked by a third-party verifier. As fallback, the original program is verified.

*Encoding Loop Abstractions into the Program.* We found one publication that also encodes loop accelerations into a modified program [23]. Here, the accelerated loop variant is added in such a way that the alternative code will be entered based on non-deterministic choice. The main motivation is to investigate how this can create synergies with invariant generation, i.e., whether existing invariant generators can be improved by also providing the results of the acceleration in the program. Compared to that, our approach is more general, as we also consider overapproximating loop abstractions. Instead of non-deterministic choice, we present an approach to determine which strategies to use automatically using CEGAR.

## 2 Preliminaries

We quickly introduce some notation and common concepts that will later be used in Sect. 3.1.

**Program Semantics.** For simplicity we will consider a programming language where the set *Ops* of possible program operations consists of simple assignments and assumptions. We represent the programs as control-flow automata (CFA). A CFA $C = \{L, l_0, G\}$ consists of a set $L$ of program locations (modeling the progam counter), an initial program location $l_0$, and a relation $G \subseteq L \times Ops \times L$ that describes the control-flow edges (each modeling the flow from one program location via a program operation to a successor program location). The concrete semantics of such a CFA is given by the (labeled) transition relation $\rightarrow \subseteq C \times G \times C$ over the set $C$ of concrete program states. We will write $c_1 \overset{g}{\rightarrow} c_2$ if the concrete state $c_2$ can be reached from $c_1$ via the control-flow edge $g \in G$.

**Program Analysis.** Our approach will work for many different kinds of program analysis. Typically, a program analysis is characterized by some abstract domain $D$ that defines a set $E$ of abstract states as well as an abstract transfer relation $\rightsquigarrow \subseteq E \times G \times E$, which determines which abstract states can be reached from the initial state $e_0 \in E$. One common way to design a program analysis is to determine the set of reachable abstract states by keeping track of a set `reached` $\subseteq E$ of already reached abstract states and a set (or list) `waitlist` $\subseteq E$ of abstract states that still need to be explored.[1]

**CEGAR.** Whenever a program analysis finds a counterexample, there are two possibilities. Either this turns out to correspond to an actual execution trace of the original program, and we have shown that the program violates the specification, or the counterexample is infeasible, meaning that it is only found because the abstraction level of the analysis is too coarse. This has led to the development of *counterexample-guided abstraction refinement*, or CEGAR for short [14]. The idea here is that one can extract information from the counterexample with which the abstract domain can be refined. For example with predicate abstraction [2], one can use the counterexample to compute predicates that —if tracked— rule out the infeasible counterexample. In order to formalize CEGAR, we will introduce the refinement operator:

$$\texttt{refine} : (\texttt{reached}, \texttt{waitlist}) \mapsto (\texttt{reached'}, \texttt{waitlist'})$$

Once an infeasible counterexample is found, the refinement operator is called with the current set of reached abstract states and the waitlist. This operator then extracts information from its inputs and returns a new set of reached states and a new waitlist which will then be used for further state-space exploration. In case the counterexample is feasible, the refinement operator will not remove the violation state(s) from the set of reached abstract states, which signals that the analysis found a bug and can terminate.

# 3 Loop Abstractions

We propose the approach of multi-strategy program analysis, which enables one tool to use several different loop-abstraction strategies simultaneously in one state-space exploration. In the following, we will first look at the theory behind loop abstractions and some practical examples for such strategies. After that, we will introduce our CEGAR refinement approach for loop abstractions in Sect. 3.2.

## 3.1 Theory

For verification, we usually use overapproximations if the goal is to find a proof of correctness. For loop control flow, such an overapproximation is called a *loop abstraction*, while precise methods are called *loop acceleration*. Whenever it is not important whether the technique is precise or overapproximating, we will just refer to the techniques as loop abstraction.

It is common to apply loop abstractions by replacing the loop statement $S$ with some alternative program statement $S'$ [1, 23]. Intuitively, it is often clear whether this will overapproximate the program behavior, but we can also formalize this

[1] In the literature, this is also know as a worklist algorithm [24]; here we will adhere to the terminology used in the Handbook of Model Checking [6].

using strongest postconditions. We write $sp(S, P)$ for the strongest postcondition of a program statement $S$ and a predicate $P$. Assume we have a program statement $S$ that contains a loop, i.e., $S =$ `while (C) do B`, where the body `B` inside $S$ may itself contain loops. For a loop abstraction, the goal is to find an alternative program statement $S'$ such that $\{P\}S\{sp(S', P)\}$ is a valid Hoare triple. If this requirement is fulfilled, then we can soundly replace $S$ by $S'$ in the program for the purpose of verification. In other words, $S'$ is an abstraction of $S$ if $sp(S, P) \Rightarrow sp(S', P)$. It is possible to find such rewriting schemes for a loop without knowing the exact form of the loop. This is best shown by two examples.

**Havoc Abstraction.** Let us look at the rather simple loop abstraction that served as example in Sect. 1, which we call *havoc abstraction*. Here we replace the loop `while C do B` by a havoc statement `havoc(mod(B))` that is guarded in such a way to ensure it is only executed if the loop condition holds, and after it is executed, the loop condition does not hold anymore. The havoc statement discards any information about the values of a set of variables. Here we use the set `mod(B)` of variables that are modified in the loop body `B`. We denote the strongest postcondition of this havoc statement by $H_{B,P} = sp(\texttt{havoc(mod(B))}, P)$. We can easily prove soundness of the havoc abstraction by establishing that $H_{B,P}$ is actually a loop invariant and therefore the Hoare triple $\{P\}$ `while C do B` $\{H_{B,P} \land \neg C\}$ holds.[2]

It is obvious that we can find an alternative statement $S'$ for the while-loop that has the same post condition:

$$sp(\texttt{havoc(mod(B));assume(!C)}, P) = H_{B,P} \land \neg C$$

We therefore have found a statement whose strongest post is an overapproximation of the strongest post of the while loop.

**Naive Abstraction.** Another way to abstract a loop is the so-called naive loop abstraction [16]. An application to the example program from Fig. 1a is shown in Fig. 1d. Here one assigns non-deterministic values to all the variables that are modified in the loop (provided the loop condition holds when reaching the loop). Then the loop body is executed once, after which the negated loop condition is added as assumption. This essentially encodes the information that if the loop was entered, there is a "last" iteration of the loop after which the loop condition does not hold anymore and the loop therefore terminates. This is overapproximating the behavior of the original loop, since a loop, in general, is not guaranteed to terminate. From the Hoare proof of the naive abstraction, we get that $sp(B, C \land H_{B,P}) \lor P$ is an invariant of the while loop.[3]

The postcondition $(sp(B, C \land H_{B,P}) \lor P) \land \neg C$ that is shown in the proof is also the post condition of the alternative code for the loop described above:

$$\begin{aligned} &sp(\texttt{if C then \{havoc(mod(B));assume(C);B;assume(!C)\}}, P) = \\ &\qquad\qquad\qquad\qquad (sp(B, C \land H_{B,P}) \lor P) \land \neg C \end{aligned}$$

[2] Proof can be found at: https://www.sosy-lab.org/research/loop-abstraction/

[3] Proof can be found at: https://www.sosy-lab.org/research/loop-abstraction/

**Observations.** We can make three interesting observations by looking at these proofs. Firstly, we eliminated the outermost loop from the statement $S$, at the cost of overapproximation. If this can be achieved iteratively until no loops are left, the resulting overapproximation can be quickly checked by a (possibly bounded) model checker, as no loops need to be unrolled anymore.

Secondly, in the proof we actually used an invariant for applying the while-rule. Every loop-abstraction strategy can therefore be seen as a way to generate invariants for a loop based on some structural properties of the loop. In the example of the havoc abstraction, we used the fact that for a precondition $P$, $H_{B,P}$ is always preserved by a loop (provided there is no aliasing). The invariant depends on the precondition $P$, so for every precondition with which the loop can be reached, the loop abstraction yields a different state invariant. Without knowing $P$ it can only be expressed as a transition invariant that may refer to the "old" values of variables before entering the loop. One can compute a state invariant by assuming the most general precondition $P = true$, but this will often eliminate most of the useful information from the invariant. As transition invariants can often be expressed precisely by program statements, this explains why for loop abstraction, we choose to replace the loop statement with alternative program statements that capture the corresponding transition invariant. This invariant view on loop abstraction works in both ways, meaning that if an invariant is provided for a loop, we can use this invariant for abstracting the loop. It is even possible to construct an inductive proof this way, i.e., transforming the loop in such a way that model checking of the resulting program will essentially carry out a combined-case (k-)inductive proof [15].

The third observation is that the invariant of one loop abstraction might sometimes imply the invariant of another loop abstraction. This is the case in the two examples: the invariant for havoc loop abstraction is implied by the invariant we use in the naive loop abstraction. This means we can build a hierarchy, where naive loop abstraction overapproximates the original loop, and havoc abstraction overapproximates naive abstraction. We will exploit the idea of this abstraction hierarchy later in Sect. 3.2 for an abstraction-refinement scheme.

**Constant Extrapolation.** For loops where we can calculate the exact number of iterations as well as the final values of the variables assigned in the loop (e.g., because the loop is linear or otherwise easily summarizable) we can simply accelerate the loop by replacing it with assignment statements for the final variable values. The application of constant extrapolation to the program from Fig. 1a is shown in Fig. 1c. For the program in Fig. 2, this would replace the loop with a statement that increments the variable `i` by `N`. For programs like the one shown in Fig. 3 that contains a potential property violation inside the loop, one has to be careful to preserve those violations that can occur in any of the loop iterations.

### 3.2 Combining Strategies for Loop Abstraction

In Sect. 3.1 we already introduced various ways to abstract loops, which we will in the following refer to as strategies. Intuitively, a strategy is a way to compute an abstraction of a loop that is helpful to verify a program.

Since there are often many different strategies that could be applied to a loop in the program, we need to make some choice about which strategies to use. The simplest approach that is used in the state-of-the-art verification tool VeriAbs is to choose the most promising that can be applied for each loop, generate a program where the loops are rewritten according to these strategies, and hand this program over to a (possibly bounded) verifier for verification.

This has the downside that in cases where the program contains multiple loops, the chosen approximations might be either not abstract enough for the verifier to calculate the proof efficiently or too abstract, such that the proof of the property does not succeed. Choosing a good abstraction level is one of the key challenges in software verification. One successful way how this can be solved is counterexample-guided abstraction refinement (CEGAR) [14].

Our idea is therefore to use CEGAR in order to refine the abstraction of the program dynamically during the program analysis, which allows us to try multiple strategies for the same loop in the program, and even different strategies for the same loop at different locations in the state-space exploration. Because a program analysis operates on the CFA, and loop abstractions correspond to transition invariants that can often be expressed naturally as a sequence of progam instructions, we choose to encode the loop abstractions directly into the CFA. This allows us to realize the CEGAR approach for loop abstractions independently of the details of the exact program analysis that is used.

**Encoding of Strategies.** We encode strategies that are to be considered directly into the CFA of the program. The CFA for a program statement $S$ such as a loop has a unique entry node $\alpha$ and a unique exit node $\omega$. The application of a strategy to this statement results in the statement $S'$ and a CFA with an entry node $\alpha'$ and an exit node $\omega'$. We attach the CFA for the statement $S'$ of a strategy with two dummy transitions $\alpha \rightarrow \alpha'$ and $\omega' \rightarrow \omega$, as depicted in Fig. 4. Here, we explicitly denoted the entry edge for the strategy application with the keyword `enter` followed by an identifier that makes clear which strategy was applied (here, `h` stands for havoc). The resemblance to function call and return edges is not a coincidence. By keeping track of the currently entered strategy applications, e.g. in form of a stack, it will always be clear which parts of the execution trace correspond to executions in the original program, and which parts are part of some —potentially overapproximating— strategy application. For nested loops, we can apply the strategies starting from the inner-most loop and construct alternatives in the CFA for all possible strategy combinations.

A CFA that is augmented with strategies in this way contains all program traces of the original program, and can non-deterministically branch into any of the strategy applications. In order to make use of this modified CFA, the analysis needs to be able to distinguish between the original control flow and nodes in the CFA at which we start to apply a particular strategy. The important nodes for this are the entry nodes for each of the strategy applications, so we augment the modified CFA $C = (L, l_{init}, G)$ with a strategy map $\sigma : L \rightarrow N$ that maps each CFA node $l \in L$ to a strategy identifier $\sigma(l) \in N$ and call the resulting tuple $\Gamma = (C, \sigma)$ a *strategy-augmented CFA*. The set $N$ of strategy identifiers

```
void main() {
  int i = 0;
  while (i<N) {
    i=i+1;
  }
  assert (i==N);
}
```

Fig. 2: Example program 1: potential property violation outside the loop

```
void main() {
  int i = 0;
  while (i<N) {
    i=i+2;
    assert(i%2==0);
  }
}
```

Fig. 3: Example program 2: potential property violation inside the loop

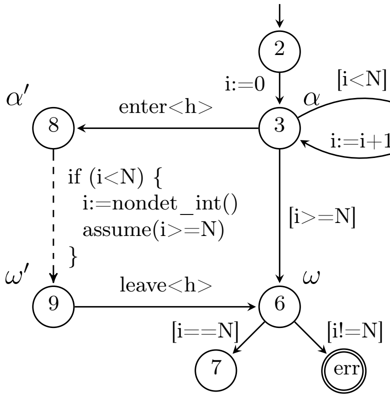


Fig. 4: CFA $C$ of example program from Fig. 2, with an additional application of the havoc strategy

contains a special strategy $b$, which we call the base strategy. The strategy map $\sigma$ maps the entry node for each strategy application to the corresponding strategy's identifier, while all other nodes are mapped to the base strategy $b$.

In a program analysis, we can now use the strategy map for selecting exactly the transitions we want to follow. For example, we can always follow the original program by excluding all transitions to CFA nodes with an associated strategy identifier that is different from the base strategy. By using a more general selection function, we have fine-grained control over which strategies we are applying, which we will describe in the following. As this modifies only the transition relation of the state-space exploration, it can be seamlessly applied to a wide variety of such algorithms.

**Selection of Strategies.** At any node $l$ in an augmented CFA, we can calculate the set $A \subseteq N$ of available strategies as:

$$A = \{\sigma(l') \mid \exists g \in G : l \xrightarrow{g} l'\}$$

In order to define which strategies should be applied (e.g., because others overapproximate too much and lead to false alarms), we define a precision set $\pi_S \subseteq N$

which we call the *strategy precision*. This precision can be tracked along each abstract state of the program analysis. In practice this precision is tracked for each program location separately, but for simplicity of presentation, we will only consider a global precision here. Semantically the precision expresses which strategies are allowed to be taken from the current abstract state. We can now express different selection approaches by defining a function $\texttt{select} : \mathcal{P}(N) \times \mathcal{P}(N) \to \mathcal{P}(N)$, which needs to fulfill the property $\texttt{select}(A, \pi_S) \subseteq A \cap \pi_S$.

The exact choice of the function `select` depends on the use case and the set of available strategies. One possibility which we will use is to define a partial order $\sqsubseteq$ over the set of available strategy identifiers, and derive the selection function in the following way:

$$\texttt{select}(A, \pi_S) = \{s \in A \cap \pi_s \mid \nexists s' \in A \cap \pi_s : s \sqsubseteq s'\} \tag{1}$$

Such a partial order can be based on the invariant hierarchy of the loop-abstraction strategies, as motivated in Sect. 3.1. It is of course not guaranteed that deciding whether one invariant implies the other is actually decidable. But depending on the strategies considered, one can also just take some design decisions regarding the partial order. In general it is desirable to have the base strategy as greatest lower bound, since as long as only overapproximation is considered, this is the most precise strategy.

The selection function above will return the most abstract strategies, i.e., that overapproximate most. Once we rule those out by removing their strategy identifier from the precision, more and more precise strategies will be returned.

**CEGAR Refinement Chaining.** We can now define the refinement operator `refine` for precision-based loop acceleration on top of any refiner of an existing analysis, which we will call the wrapped refiner $\texttt{refine}_{\mathbb{W}}$. This can be done by composing the refinement operator $\texttt{refine}_{\mathbb{W}}$ with the strategy-refinement operator $\texttt{refine}_{\mathbb{S}}$, which updates the strategy precision with information from the error path:

$$\texttt{refine} = \texttt{refine}_{\mathbb{S}} \circ \texttt{refine}_{\mathbb{W}} \tag{2}$$

Since the wrapped refinement operator is executed first, it gets the possibility to remove all error states from the reached set, in which case $\texttt{refine}_{\mathbb{S}}$ has nothing to do and will just return its inputs. If there are still error states left in the reached set after $\texttt{refine}_{\mathbb{W}}$ was executed, this means that the inner refinement has discovered a feasible error path for the augmented CFA. Now it depends on whether any overapproximating strategies were used on the error paths that are present in the reached set. If there are none, then the error path is indeed also present in the real program and $\texttt{refine}_{\mathbb{S}}$ returns the reached set with the error state(s), indicating that a bug has been found. An example for this would be the case where only constant extrapolation has been used along the path. If there are overapproximating strategies such as the havoc abstraction on an error path, we can adapt the strategy precision in order to rule out that we will find the same error path again after the refinement. For that, we locate the first abstract state on the path whose successor enters an overapproximating strategy (the so-called pivot state) and adapt the strategy precision such that this strategy can not be selected in the future. We then remove all (transitive) successor abstract states of that pivot state from the set of reached abstract states.

**Example.** The chaining of the refinement operators is best visualized by looking at an example. Using the running example from Fig. 2, we can look at the key steps in the CEGAR refinement. Let us assume we are only using the havoc strategy, i.e., the augmented CFA will look like shown in Fig. 4. Based on this CFA, an example for how a generic state-space exploration could look like is depicted in Fig. 5. In Fig. 5a we start at an abstract state with three components. The first one encodes the program location and is set to 2, since program location 2 is the initial progam location in the CFA. Component $e_0$ encodes the analysis-specific domain part of the abstract state, e.g., for predicate abstraction this could be a set of predicates. The last component is the strategy precision. It contains the base strategy ($b$) as well as the havoc strategy ($h$). From this state, the state-space exploration continues to program location 3, where the selection of strategies in the transition relation only allows us to proceed into the application of the havoc abstraction. From there, we eventually reach the error location.

This is where the CEGAR refinement operator is first called. Since the path formula to the error location is actually feasible, the wrapped refinement operator return the inputs unchanged, and our strategy refinement operator takes over. Here we discover that an overapproximating strategy was used on the path. We update the strategy precision of the second state (the one at program location 3) such that the havoc strategy cannot be chosen anymore. We then remove all successors of the pivot state from the set of reached abstract states (and the waitlist), add the modified state to the waitlist, and return both sets.

The resulting reachability graph will look like in Fig. 5b. From there, the state-space exploration can continue as shown in Fig. 5c. We again discover an error path, this time however the wrapped refinement operator can determine that this error path is infeasible. In case of a predicate abstraction, a predicate like $i < N$ would be discovered and added to the predicate precision of $e_1'$ at program location 3. All successors after location 3 are removed again and the wrapped refinement operator returns. Since there is no error state present anymore in the set of reached states, the strategy refinement operator returns its inputs unchanged. The state-space exploration then continues by adding a new abstract state for program location 4 and so on, as depicted in Fig. 5d.

**Transformation into Source Code.** We also provide functionality to convert the loop abstractions we found back into source code, such that our findings can be used and validated by others. For that, we provide two different mechanisms. The first is that whenever we are able to generate a proof using some loop-abstraction strategy, we generate a modified version of the input program where just the loops are changed to reflect the effect of the loop abstraction. The second mechanism is that we provide a way to analyze a C program such that for each loop in the program and each loop-abstraction strategy, we create a patch file for the program (in case the strategy is applicable) that —when applied— will apply the loop abstraction on the source-code level.

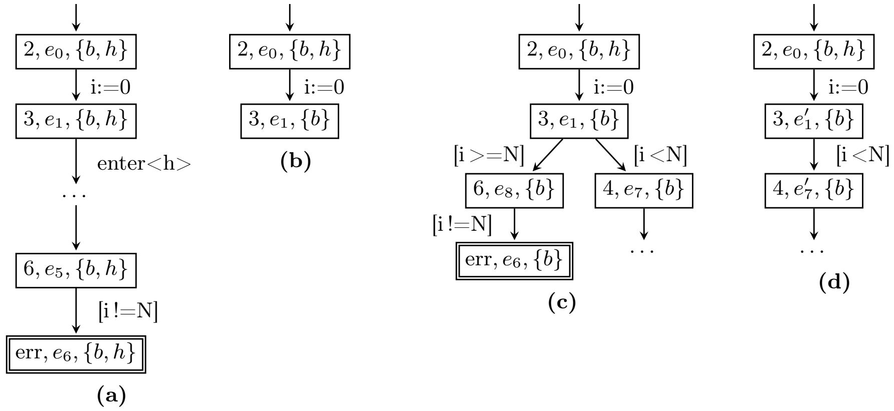


Fig. 5: Example for constructing a reachability graph of a program analysis on Fig. 4 using chained CEGAR refinements: **(a)** initial ARG until first refinement, **(b)** strategy precision updated after refinement (strategy $h$ removed from precision), **(c)** state-space exploration on the original program continues, **(d)** exploration continues after a regular CEGAR refinement ($e_1$ replaced by $e'_1$)

## 4 Evaluation

As a first step, we implemented the three loop-abstraction strategies that we described in Sect. 3.1 into the state-of-the-art verification framework CPAchecker: havoc abstraction ($h$), naive abstraction ($n$), and constant extrapolation ($c$). In addition, we also implemented so-called output abstraction ($o$) [15]. For the evaluation, we define the following (partial) order on which the function `select` will be based:

$$b \sqsubseteq o \sqsubseteq c \sqsubseteq n \sqsubseteq h \tag{3}$$

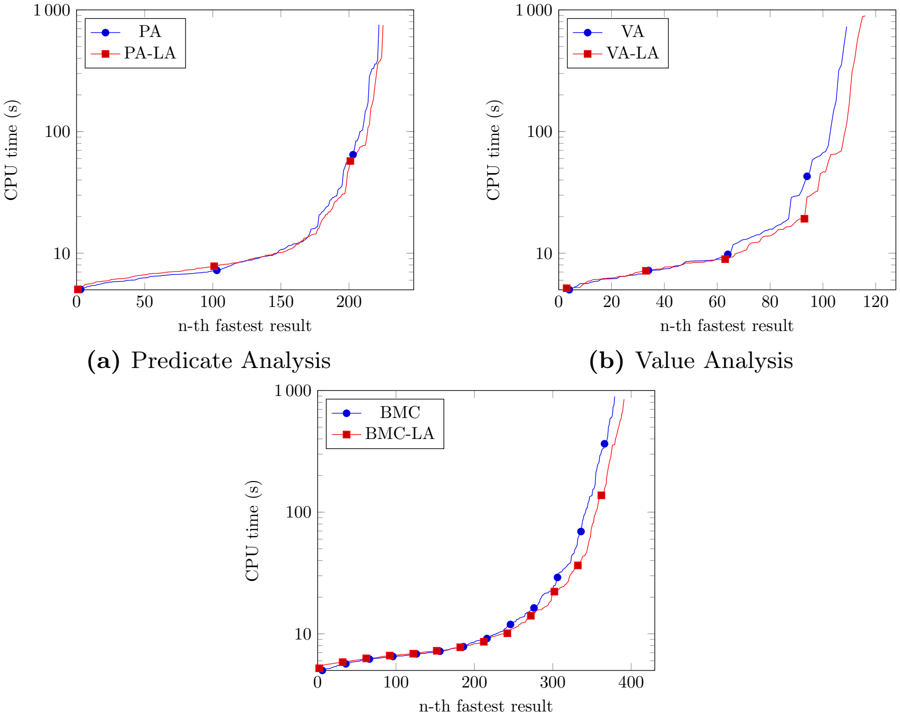
We are interested in answering the following research questions:

- **RQ1:** Can our CEGAR-style loop-abstraction scheme soundly improve a verifier like CPAchecker independently of the underlying analysis?
- **RQ2:** Are these abstractions also useful for other verifiers?

We conduct an experiment for each RQ in Sect. 4.2 to abtain answers.

### 4.1 Benchmark Environment

For conducting our evaluation, we use BenchExec to ensure reliable benchmarking [12]. All benchmarks are performed on machines with an Intel Xeon E5-1230 CPU (4 physical cores with 2 processing units each), 33 GB of RAM, and running Ubuntu 20.04 as operating system. All benchmarks are executed with resources limited to 900 s of CPU time, 15 GB of memory, and 1 physical core (2 processing units).



**(a)** Predicate Analysis

**(b)** Value Analysis

**(c)** BMC Analysis

Fig. 6: Quantile plots comparing performance of plain analyses with their versions that use loop-abstraction strategies; only correct results are considered

### 4.2 Experiments

For our experiments we use verification tasks taken from the SV-Benchmarks set of SV-COMP 2022 [3, 4]. Here we selected only the 765 reachability tasks from the subcategory ReachSafety-Loops, as these cover a wide range of interesting loop constructs while at the same time only using a limited set of features of the programming language C, which allows us to focus on the algorithms instead of having to deal with lots of special cases.

**RQ1.** In a first experiment, we evaluate whether our approach can improve the overall results, and whether our new framework introduces significant overhead, for three analyses of CPAchecker: (1) predicate analysis (PA) [5] configured to use predicate abstraction [7, 9, 18], (2) value analysis (VA) [6, 11], which is an extension of constant propagation [21], and (3) predicate analysis configured to work as bounded model checking (BMC) [13]. For improvements we will look at effectiveness as well as efficiency. By effectiveness we mean an increase in the number of solved verification tasks while at the same time preserving soundness of the results, i.e., no increase of the number of wrong proofs or wrong alarms. For efficiency we will take a look at how our approach affects the verification time of successfully verified tasks.

Table 1: Results for predicate abstraction (PA), value analysis (VA), and bounded model checking (BMC), without vs. with loop abstraction (LA)

| | PA | PA-LA | VA | VA-LA | BMC | BMC-LA |
|---|---|---|---|---|---|---|
| Total | 765 | 765 | 765 | 765 | 765 | 765 |
| Total proofs | 533 | 533 | 533 | 533 | 533 | 533 |
| Correct proofs | 164 | 163 | 33 | 35 | 235 | 248 |
| Incorrect proofs | 0 | 0 | 0 | 0 | 0 | 0 |
| Total alarms | 232 | 232 | 232 | 232 | 232 | 232 |
| Correct alarms | 58 | 62 | 76 | 81 | 144 | 144 |
| Incorrect alarms | 0 | 0 | 0 | 0 | 0 | 0 |

Table 2: Impact of loop abstractions on solving capabilities of the software verifiers UAutomizer (UA), Cbmc, and Symbiotic, without vs. with loop abstraction (LA) via generated abstracted programs

| | UAutomizer | UAutomizer-LA | Cbmc | Cbmc-LA | Symbiotic | Symbiotic-LA |
|---|---|---|---|---|---|---|
| Total | 18 | 18 | 18 | 18 | 18 | 18 |
| Total proofs | 14 | 14 | 14 | 14 | 14 | 14 |
| Correct proofs | 12 | 13 | 0 | 13 | 12 | 13 |
| Incorrect proofs | 0 | 1 | 0 | 1 | 0 | 1 |
| Total alarms | 4 | 4 | 4 | 4 | 4 | 4 |
| Correct alarms | 0 | 3 | 1 | 3 | 1 | 3 |
| Incorrect alarms | 0 | 0 | 0 | 0 | 0 | 0 |

The quantile plots in Fig. 6 show that we are able to slightly improve the results for all analyses. Both effectiveness and efficiency is improved, and thus, there is no noticeable overhead. We use PA-LA, VA-LA, and BMC-LA to refer to the variants of the analyses that use our CEGAR-style loop-abstraction scheme. As expected, the overhead of applying loop abstraction in cases where this does not help with solving the verification task does not add a significant overhead to the verification time. Table 1 shows that our approach is also sound, i.e., it does not increase the number of incorrect results.

Another observation is that there are more proofs as well as property violations found this way. The latter is possible because constant extrapolation is a precise abstraction, meaning that a counterexample found using this strategy corresponds to a feasible error path in the program.

The experimental data so far suggests that if loop abstraction helps with verification, the verification will usually succeed very quickly. For all tasks where the verdict improves, the application of loop abstraction reduces the verification time from a timeout, i.e., more than 900 seconds, to less than 10 seconds. On closer inspection, we find a total of 18 verification tasks where the loop abstraction is essential in proving the program correct with the used analyses. When comparing the different analyses, the effect is most noticeable with bounded model checking, which is not surprising given the fact that BMC alone can not prove programs with unbounded loops. There are 6 tasks where predicate analysis improved, 5 tasks for value analysis, and 17 tasks for BMC.[4] Since our framework supports exporting the accelerated loops into the source code, we can use the 18 abstracted programs that improved CPAchecker's results in the next experiment, where we check whether these are also useful for other software verifiers.

**RQ2.** In the second experiment we take a look at whether our approach has the potential to improve the results of other state-of-the-art verification tools as well. In order to be able to do so without having to modify the existing tools, we take those programs where loop-abstraction strategies were able to improve the results for CPAchecker and automatically generate the abstracted programs that can then be fed to other verifiers. In our case, we use the three well-known verifiers Cbmc, Symbiotic, and UAutomizer.

The results of all three verifiers improve if loop abstraction is applied, as shown in Table 2. The table shows the results for the verifiers on the original verification tasks (columns without suffix LA) and on the abstracted programs (column with suffix LA). Note that this will not be the case in general, but for the selected verification tasks, we know that one of our implemented loop abstraction strategies is actually sufficient to prove the program correct. In general, if a loop abstraction over-approximates too much, the verifier will quickly find an error path, in which case we would execute the verifier on the original program. There is also one program for which our loop abstraction leads to a wrong proof, which is due to a bug in our translation back into source code.

The main observation here regarding our research question is that the results of all three verifiers can be improved by applying loop abstraction. We get the largest improvement for the bounded model checker Cbmc. This is not surprising and in line with the results from the bounded model checking with CPAchecker.

## 5 Conclusion

Loop abstraction is a technique for program verification that is currently not used by many of the state-of-the-art verification tools. In our experiments we have shown that mature verifiers can still benefit even from very simple loop abstractions. By adding more sophisticated loop-abstraction strategies in the future, we hope to achieve even better results that further improve the state-of-the-art. We make the loop abstractions that we implemented available to other tools by generating modified versions of the input programs, such that also other tools can benefit from loop abstractions in the future.

In this paper, we have also addressed the problem of how to select the right combination of loop abstractions for programs with multiple loops. Instead of deciding upfront which combination to choose, we use a novel approach based on CEGAR to automatically refine the loop abstractions as the analysis progresses. By using the control flow as interface for program analyses, we are able to apply our approach to a wide range of existing analyses and abstract domains, without additional implementation overhead.

[4] Detailed results at: https://www.sosy-lab.org/research/loop-abstraction/

**Data-Availability Statement.** The software and programs that we used for our experiments, including the generated programs with abstracted loops, are open source and available on our supplementary web page at https://www.sosy-lab.org/research/loop-abstraction/ and in the reproduction package at Zenodo [10].

**Funding Statement.** This project was funded in part by the Deutsche Forschungsgemeinschaft (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 (2019). https://doi.org/10.1109/ASE.2019.00121
2. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proc. PLDI. pp. 203–213. ACM (2001). https://doi.org/10.1145/378795.378846
3. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
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., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
6. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
7. 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). https://doi.org/10.1007/s10009-007-0044-z
8. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE. pp. 29–38. IEEE (2008). https://doi.org/10.1109/ASE.2008.13
9. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010)
10. Beyer, D., Lingsch Rosenfeld, M., Spiessl, M.: Reproduction package for SEFM 2022 article 'A unifying approach for control-flow-based loop abstraction'. Zenodo (2022). https://doi.org/10.5281/zenodo.6793834
11. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11
12. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
13. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14

14. 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). https://doi.org/10.1145/876638.876643
15. Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: Proc. DATE. pp. 1407–1412. IEEE (2015). https://doi.org/10.7873/DATE.2015.0245
16. Darke, P., Khanzode, M., Nair, A., Shrotri, U., Venkatesh, R.: Precise analysis of large industry code. In: Proc. APSEC. pp. 306–309. IEEE (2012). https://doi.org/10.1109/APSEC.2012.97
17. Frohn, F.: A calculus for modular loop acceleration. In: Proc. TACAS (1). pp. 58–76. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_4
18. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV. pp. 72–83. LNCS 1254, Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
19. Jeannet, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. In: Proc. POPL. pp. 529–540. ACM (2014). https://doi.org/10.1145/2535838.2535843
20. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_15
21. Kildall, G.A.: A unified approach to global program optimization. In: Proc. POPL. pp. 194–206. ACM (1973). https://doi.org/10.1145/512927.512945
22. Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property checking array programs using loop shrinking. In: Proc. TACAS (1). pp. 213–231. LNCS 10805, Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_12
23. Madhukar, K., Wachter, B., Kröning, D., Lewis, M., Srivas, M.K.: Accelerating invariant generation. In: Proc. FMCAD. pp. 105–111. IEEE (2015)
24. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999). https://doi.org/10.1007/978-3-662-03811-6
25. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)
26. Silverman, J., Kincaid, Z.: Loop summarization with rational vector addition systems. In: Proc. CAV, Part 2. pp. 97–115. LNCS 11562, Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_7