






TRANSVER: A Modular Program-Transformation Framework for Reduction to Reachability

Dirk Beyer^{}, Marek Jankola^{}, Marian Lingsch-Rosenfeld^{},
Tian Xian^{}, and Xiyue Zheng^{}

LMU Munich, Munich, Germany

Abstract. Software verification is a complex problem, and verification tools need significant tuning to achieve high performance. Due to this, many verifiers choose to specialize on basic reachability properties. Instead of implementing algorithms for each possible specification, some verifiers implement known transformations from the given specification to reachability on their internal representations. Unfortunately, those internal transformations are not reusable by others. To improve this situation, we propose TRANSVER, a tool which offers transformations as modular stand-alone component, modifying the input program instead of the internal representation, enabling their usage as a preprocessing step by other verifiers. This way, we separate two concerns: improving the performance of reachability analyses and implementing efficient transformations of arbitrary specifications to reachability. We implement the transformations in a framework that is based on *instrumentation automata*, inspired by the BLAST query language. In our initial study, we support three important concrete specifications for C programs: *termination*, *no-overflow*, and *memory cleanup*. We conduct experiments with ten different verifiers. The experiments evaluate the efficiency and effectiveness of our transformations. The results are promising: Our transformations can extend existing verifiers to be effective on specifications for which they have no integrated support, and the efficiency is often similar or better to state-of-the-art verifiers that have integrated support for the considered specifications.

Keywords: Specification · Monitoring · Specification Reduction · Reachability · Formal Verification · Model Checking · Software Verification · Program Analysis

1 Introduction

Software verification is the problem to decide, for a given program P and specification φ , whether the program P satisfies its specification φ , in short: $P \models \varphi$. To address this issue, a software verifier is usually constructed in one of two ways: (a) implement a verification algorithm for $P \models \varphi$ or (b) *reduce* the problem by transformation in order to solve it using an existing algorithm. The transformation-based approach consists of two steps and assumes an existing verifier v that supports a specification φ' . In the first step, the problem $P \models \varphi$ is transformed to a problem $P' \models \varphi'$, such that $P \models \varphi$ holds if and only if $P' \models \varphi'$ holds. In the second step, the problem $P' \models \varphi'$ is solved by the existing verifier v .

Several verification tools already choose to use transformations [1, 2, 3], since transformations allow for a separation of concerns: (1) support a rich set of specifications and (2) tune the performance of specialized algorithms for one particular specification. For example, given a verifier that supports reachability, the verifier can be extended to support other specifications, like termination and no-overflow, for which a transformation to reachability is available.

Currently, a developer who desires to extend a verifier to support more specifications using the transformation approach has to re-implement the transformation. The goal of this paper is to show (a) that it is possible to construct verifiers in a *modular* way from independent components, that is, compose an ‘off-the-shelf’ transformation with an ‘off-the-shelf’ verifier, such that a transformation can be used with arbitrary verifiers for C programs to support more specifications, (b) that transformation-based approaches can be even more efficient than integrated support for specifications, and (c) that the standalone transformations do not necessarily lead to a performance decrease in comparison to tool-specific implementations for checking the input specification (using internal transformation).

To achieve this, we developed **TRANSVER**, a transformation framework and tool for software verification, in which verification engineers write specifications at a high-level as *instrumentation automata* (IA), which contain instructions to instrument and monitor the program. Instrumentation automata are inspired by the BLAST query language [4] and SLIC [5], which is a specification language for SLAM [6]. Both specification languages are based on monitor automata and have shown to be useful in practice, since the convenient and succinct notation of monitor automata is often easier to understand than LTL formulas. Monitor automata can be implemented either by instrumentation of the monitor into the program code [4, 5] or by an implicit on-the-fly product construction in which the monitor is a separate analysis component [7, 8]. Instrumentation automata observe the *control-flow automaton* of a program, and weave monitoring instructions and error assertions into the control-flow where appropriate.

To showcase our approach, we have implemented three transformations that reduce verification problems for which the specification is to check *termination*, *memory cleanup*, or (arithmetic) *no-overflow*, to verification problems for which the specification is to check reachability. Those three kinds of specifications are important and often used, because we want programs to not cycle infinitely but progress with useful computation (termination), we want the programs to free all allocated memory such that we can use the programs as components as part of other programs (memory cleanup), and we want the software to not run into undefined behavior, like signed-integer overflow, and always have values within their types (no-overflow). The target specification is reachability since most verifiers (28 verifiers) for software verification that participated in the competition on software verification (SV-COMP) in 2024 [9] support it and have focused on tuning their algorithms towards best performance on such tasks. Fewer verifiers support other specifications, such as *no-overflow* (19 verifiers), *termination* (16 verifiers), or *memory cleanup* (21 verifiers), since adding support for multiple properties is a time-consuming process.

In summary, the advantage of our approach is a clear separation of concerns, because the concern of optimizing a verification algorithm (for reachability) is now completely independent from checking whether a (non-reachability) specification is satisfied. In addition, this approach opens new opportunities, for example, using test-generation tools like fuzzers to check for violations of a specification defined as instrumentation automaton. For example, it is now possible to construct a fuzzing-based non-termination checker similar to existing tools [10, 11] without spending any development effort on the fuzzing tool. All it takes is developing an instrumentation automaton for TRANSVER. In particular, the verifiers EMERGENTHETA [12], THETA [13], and THORN used TRANSVER as a preprocessing step for SV-COMP 2025, which demonstrates TRANSVER’s modularity.

Contributions. This paper makes the following contributions:¹

- We propose a verifier-independent, modular transformation framework for the instrumentation of C programs. It allows verifiers for reachability to be used also for other specifications as well.
- We provide an open-source implementation of our transformation framework.
- We conduct an experimental evaluation on the SV-COMP benchmark set of C programs, which shows that we can effectively extend existing verifiers to specifications that they did not support before (RQ 1), that verifiers combined with transformations sometimes even outperform verifiers specialized in verifying the original property (RQ 2, RQ 3), and that verifiers with internal transformations are not necessarily more efficient than a composition using our transformation framework and an ‘off-the-shelf’ verifier (RQ 4).

2 Related Work

Program transformations have a wide variety of applications [15, 16]. We focus on three kinds [17] of program transformations closely related to our approach.

Reducers can simplify complicated language constructs, for example, by sequentializing concurrent programs [18, 19], by reducing the program to a simplified syntax [1, 20], or by merging multiple loops into one single loop [21, 22]. There are also reducers that replace program constructs (for example, loops) by constructs that are easier to verify [23, 24, 25, 26, 27, 28]. Sometimes they even use information from run-time verification to ease the static analysis [29].

Specifications transformers convert a problem $P \models \varphi$ to a new problem $P' \models \varphi'$. This makes it possible to use algorithms for the verification of φ' to also verify φ [2, 30, 31, 32, 33, 34, 35, 36, 37]. Specifications transformers can also be used for testing, in order to transform a program and a coverage specification to another program and coverage specification, such that existing tools for test generation or test-suite analysis can be used [38, 39]. Our work focuses on this kind of program transformation. We improve over existing works by two aspects: modularity and generality. Outputting a modified C program makes the application of any C verifier that supports reachability effortless. Moreover, we demonstrate that our framework supports transformations for multiple properties.

¹ A preliminary version of this article was published as technical report [14].

Instrumentors add code to the program that is used to collect information for further analysis. Some instrumentors express the verification goal as part of the program to be verified. This has been studied for a variety of applications. One such example is in the context of verification witnesses [40, 41] in the case of MetaVal [42], which creates a product of the witness and the program. Another example is proof-carrying code [43], where the proof is embedded into the program. Furthermore, instrumenting additional logic into the program allows for its run-time monitoring [44, 45, 46] or improves the verification process, for example by using shadow memory [47, 48]. Furthermore, there also exist configurable instrumentors, for example, tools like AspectJ [49] and AspectC++ [50] are used in aspect-oriented programming. Moreover, there are verifiers with configurable instrumentation engines that can produce instrumented code. For example, SYMBIOTIC [51, 52, 53] can instrument LLVM and ESBMC [54, 55] can instrument C code to unroll loops or inject additional goals.

Instrumentation can also speed up the verification of programs containing operations over arrays using ghost variables and rewriting rules [56]. The three main differences between our and the mentioned approach are (a) that we transform the task from other specifications to reachability specifications, and the above-mentioned approach transforms programs with extended quantifiers in assertions to programs with simpler assertions, (b) the implicit ordering of instrumentation, and (c) our approach does the matching on a control-flow automaton of the program and not on its syntax.

3 Background

Control-Flow Automata. We model the control-flow of a program as *control-flow automaton (CFA)* [57]. A CFA (L, l_0, G) consists of a finite set L of locations, an initial location $l_0 \in L$, and a set $G \subseteq L \times Ops \times L$ of edges, which represent that the control flows from one location to the next while executing an operation. We use the special function `nondet()`, which returns a non-deterministic value, and the special operation `assert(π)`, which means that the condition π over program variables holds whenever the program execution reaches this operation (condition π is a location invariant). Figure 1 shows an example of a program and a corresponding CFA. The program-to-CFA transformation is implemented in the software-verification framework CPACHECKER [8], which we use in TRANSVER as a component. CPACHECKER also stores information about the type of variables and expressions used on the edges of the CFA. We assume that the CFA is constructed from a C program such that every edge contains at most one operation, and at most one arithmetic or boolean expression (composite expressions are decomposed into multiple edges). Since CPACHECKER leaves expressions with multiple arithmetic expressions, e.g., $x + y + z$, during the transformation, we introduce new edges on demand to split the arithmetic operation into, e.g., `tmp = x + y` and `tmp + z`. However, instead of creating explicit auxiliary variables, we keep track of which subexpressions should be substituted in their place. A *program state* is a mapping from program variables to their values, including a program counter `pc`, which is a variable that is mapped to the current program location.

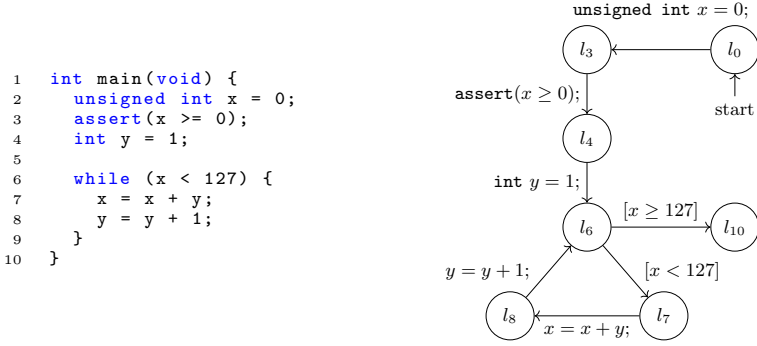


Fig. 1: Example program (left) with corresponding CFA (right)

Table 1: Specifications considered in this work, as defined by SV-COMP

Specification	Explanation
<i>reachability</i>	The function <code>reach_error</code> is not called in any execution of the program. We write <code>assert(π)</code> for <code>if (!π) reach_error()</code> .
<i>no-overflow</i>	No execution of the program produces during an operation a signed integer value that is outside the range of the signed C type.
<i>termination</i>	No execution of the program has infinitely many operations.
<i>memory cleanup</i>	No execution of the program allocates a pointer and then terminates without freeing it.

Specifications. One of the most prominent specification languages for behavioral properties is linear-time temporal logic (LTL) [58]. In the International Competition on Software Verification (SV-COMP) [59], verifiers compete in verifying several practical LTL properties². Table 1 lists the specifications from SV-COMP for which we showcase transformations within our framework in Sect. 5. There are two important subclasses of LTL formulas: *safety* and *liveness*.

Safety specifications describe properties that must hold for all reachable program states on all program executions ("something bad never happens"). A violation consists of a finite execution that has a program state for which the property does not hold. In SV-COMP, the most general version of this kind of specification is *reachability*. Other safety specifications like *no-overflow* and *memory cleanup* can be expressed as reachability by applying a program transformation.

Liveness specifications describe properties that must eventually hold on every program execution ("something good eventually happens"). A violation consists of an infinite execution on which the property never holds or a finite execution that terminates before the property holds. SV-COMP uses only one liveness property, program *termination*, which we also showcase in this paper.

² <https://sv-comp.sosy-lab.org/2024/rules.php>

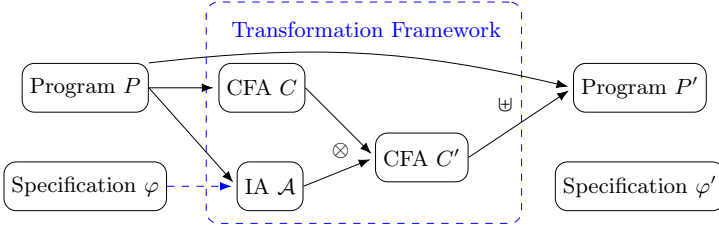


Fig. 2: Workflow of the program transformation using **TRANSVER**

4 Transformation Framework

We show the workflow of our transformation framework in Fig. 2. **TRANSVER** takes a program P and a specification φ as an input. It uses **CPACHECKER** to construct the corresponding CFA C ³ for program P and combines it with the instrumentation automaton \mathcal{A} to a modified CFA C' using the sequentialization operator \otimes . The operations of the new control-flow edges of C' , along with the corresponding line numbers at which the operations should be inserted in P , are then exported by **CPACHECKER**. Finally, the exported modifications are added to P using the instrumentation operator \uplus , in order to produce the transformed program P' . We use the CFA construction of **CPACHECKER** [3] because it supports a wide variety of C features. For a new specification, the user needs to formalize the transformation as an instrumentation automaton and construct a Java object for it inside **CPACHECKER**, marked by the blue dashed arrow in Fig. 2 from φ to \mathcal{A} . The specification can be program-specific or general for a large group of programs. We formalize the automata for three example specifications in Sect. 5.

Instrumentation Automata. An instrumentation automaton specifies how a program needs to be instrumented in order to make the original specification explicit using assertions. It is inspired by the observer automata of the BLAST query language [4]. They allow syntactic pattern matching on a given C program, followed by an action or assertion. To enable the transformation of more complex properties, we additionally match over the structure of the CFA. This makes it possible to match loops that are not immediately apparent from the program’s syntax, like `gotos`.

An *instrumentation automaton* (IA) is a tuple $(Q, q_0, Var, \delta, \alpha)$ that consists of a set Q of states, an initial state q_0 , a set Var of automaton variables, a transition relation $\delta \subseteq Q \times PATTERNS \times Ops \times \{A, B\} \times Q$, and a state-annotation function α . A transition (q, ρ, op, X, p) from a state q to a state p specifies:

- A *pattern* $\rho \in PATTERNS$ describes which C expression from a CFA edge is matched. The pattern `*` matches any symbols on the CFA edge. Furthermore, loop conditions and their negations can be matched with the special patterns `cond` and `!cond`, while the pattern `true` matches any operation. Moreover, the pattern can also contain capturing groups, and the matching results are

³ In practice, **CPACHECKER** constructs a set of CFAs, one per function. For presentation purposes, and without loss of generality, we use only one CFA in this paper.

assigned to match identifiers $\$x_0, \x_1, \dots , which can be further used like variables in the instrumented operation op . Given a C operation op^C from a CFA edge, $match(\rho, op^C)$ implies that ρ matches op^C and $vars(\rho, op^C)$ is the map of match identifiers $\$x_0, \x_1, \dots to the operands of op^C . For example, $\rho \equiv .* \$x_0 + \x_1 ; matches $op^C \equiv x + 4$, and $vars(\rho, op^C) = \{x_0 \rightarrow x, x_1 \rightarrow 4\}$.

- An operation $op \in Ops$ is a statement that can read and write to IA variables from Var , but can only read values from the match identifiers and program variables from the CFA. Moreover, it can refer to the match identifiers $\$x_0, \x_1, \dots that occur in ρ , which $sub(op, m)$ replaces with their mapped expressions from $m \equiv vars(\rho, op^C)$. E.g., for the operation op given by $sum = x_0 + x_1$; with $sum \in Var$, then $sub(op, \{x_0 \rightarrow x, x_1 \rightarrow 4\})$ results in $sum = x + 4$;
- The symbol $X \in \{A, B\}$ specifies whether the instrumented operation should be placed before (B) or after (A) the matched edge in the CFA.

The state-annotation function $\alpha : Q \rightarrow \{true, false, loop_head, init\}$ assigns to every state of the instrumentation automaton a predicate $L \rightarrow \{true, false\}$. The location predicates $true(l)$ and $false(l)$ hold for all locations and no location, respectively, $loop_head(l)$ holds for program locations l at the beginning of a loop, and $init(l)$ holds for the initial program location l .

Sequentialization Operator. The sequentialization operator \otimes takes the operations from an IA and places them in the indicated locations in an input CFA. The operator implicitly traverses both the CFA and IA in parallel (on-the-fly reduced product). During the traversal, it processes pairs of CFA locations and automaton states (l, q) , and pairs of a CFA edge $e = (l, op^C, l')$ and an automaton transition $t = (q, \rho, op, X, p)$. First, it checks whether l matches q with the predicate from the state-annotation function $\alpha(q)$ and whether ρ matches op^C . Afterwards, it instantiates the operation op , substituting the match identifiers from ρ . Lastly, it creates a new edge before or after e , depending on X . If there is no more match for e , the algorithm inserts the original edge e from the input CFA.

The sequentialization operator is implemented as [Alg. 1](#). The algorithm starts with the initialization step, which traverses the CFA and collects additional information about the program that can then be used to construct multiple concrete instrumentation automata from the input instrumentation automaton for a given program. For example, a transformation for termination initializes one automaton, cf. [Fig. 6](#), per loop in the CFA, collects all the variables used in the respective loop, and initializes one ghost variable for each. Initializing multiple automata is an optimization that users can choose to implement when initializing their defined automaton. Moreover, `initialize_automata` also returns a location from the CFA for each automaton, labeled as the initial location for the sequentialization. This optimization is used to skip parts of the CFA that are irrelevant. For example, it is used by the automaton for termination reduction to monitor one loop per automaton.

[Line 2](#) initializes the sets of locations and the edges of the output CFA. [Line 3](#) instantiates the set `waitlist` to contain the initial pair of program location and IA state, for each initialized IA from [line 1](#). The while loop starting in [line 4](#)

Algorithm 1 Sequentialization operator \otimes **Input:** a CFA $C = (L, l_0, G)$, an IA $\mathcal{A} = (Q, q_0, Var, \delta, \alpha)$ **Output:** CFA $C' = (L', l'_0, G')$

```

1:  $(\mathcal{A}_0, l_{init}^0), \dots, (\mathcal{A}_k, l_{init}^k) \leftarrow \text{initialize\_automata}(C, \mathcal{A})$ ;
2:  $L', G' \leftarrow \{l_0\}, \{\}$ ;
3:  $\text{waitlist}, \text{finished} \leftarrow \{(l_{init}^0, q_0^0), \dots, (l_{init}^k, q_0^k)\}, \{\}$ ;
4: while  $\text{waitlist} \neq \emptyset$  do
5:    $(l, q^i) \leftarrow \text{waitlist.pop}()$ ;
6:   if  $((l, q^i) \in \text{finished})$  then
7:     continue;
8:    $\text{finished.add}((l, q^i))$ ;
9:    $\text{waitlist} \leftarrow \text{waitlist} \cup \text{succ}((l, q^i))$ ;
10:  if  $\neg \alpha^i(q^i)(l) \vee \text{succ\_IA}((l, q^i)) = \{\}$  then
11:     $G' \leftarrow G' \cup \{(l, \cdot, \cdot) \in G\}$ ;
12:  else
13:     $G' \leftarrow G' \cup \text{new\_edges}((l, q^i))$ ;
14:   $L' \leftarrow L' \cup \{l^{new} \mid l^{new} \notin L' \wedge \exists l' : ((l', op, l^{new}) \in G' \vee (l^{new}, op, l') \in G')\}$ ;
15: return  $(L', l'_0, G')$ ;
```

traverses both the input CFA and the IA in parallel. States and all the other components from automaton \mathcal{A}_i , for $0 \leq i \leq k$, are marked with i in superscript.

In every iteration, it works with a pair (l, q^i) that was not yet processed. The order in which the pairs are processed is not specified. Therefore, users must ensure that the transformation given by their IA does not depend on the exploration order of the waitlist. The algorithm then computes the successors of (l, q^i) as seen in [line 9](#), if it was not yet processed. Function `succ` is defined as follows in [Eq. \(2\)](#).

$$\text{succ_IA}((l, q^i)) = \{(l, p^i) \mid \exists (l, op^C, \cdot) \in G, \exists (q^i, \rho, \cdot, \cdot, p^i) \in \delta^i : \text{match}(\rho, op^C)\} \quad (1)$$

$$\text{succ}((l, q^i)) = \begin{cases} \{(l', q^i) \mid \exists (l, \cdot, l') \in G\} & \text{if } \neg \alpha^i(q^i)(l) \\ \{(l', q^i) \mid \exists (l, \cdot, l') \in G\} & \text{if } \text{succ_IA}((l, q^i)) = \{\} \\ \text{succ_IA}((l, q^i)) & \text{otherwise} \end{cases} \quad (2)$$

To compute the successors using [Eq. \(2\)](#), we consider three cases. In the first case, the annotation of the state does not hold for the location, and in the second, none of the outgoing edges from q^i matches an outgoing edge from l . Therefore, the algorithm progresses only in the CFA and creates new pairs of q^i with all the successors of l . In the third case, the annotation $\alpha^i(q^i)(l)$ holds, and there are transitions from q^i with a pattern that matches some edge from l . This results in progressing only with the IA and pairing all IA successors of the transitions that matched some edge starting at l , as defined by function `succ_IA` in [Eq. \(1\)](#).

The condition in [line 10](#) guards the addition of new edges into the resulting CFA. If no outgoing edge is matched or the state annotation does not hold for the location, the algorithm adds all the edges from the input CFA C . Otherwise, the function `new_edges` defined in [Eq. \(4\)](#) computes all edges that instrument the original program.

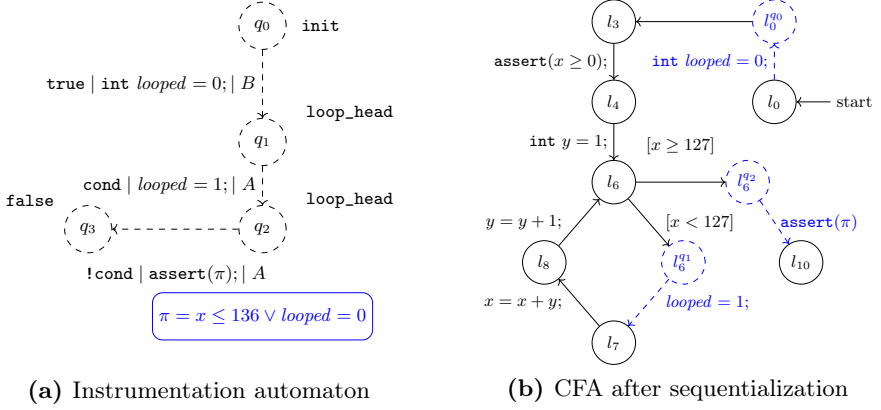


Fig. 3: Example instrumentation automaton (left) and the example CFA from Fig. 1 after sequentialization with it (right); dashed lines and locations indicate instrumented new CFA edges and locations

The function `new_edges` uses the predicate SO (*Substituted Operation*) presented in Eq. (3). The predicate is true if the operation op^{A_i} is an instantiated operation from a transition in an IA in which the match identifiers $\$x_0, \x_1, \dots were substituted by the match results (expressions) from op^C . For example, let us assume CFA edge $(l, y = z + 42; , l')$ and an IA transition $(q, .* \$x_1 + \$x_2, \text{assert}(x_1 > x_2), B, q')$, then it is the case that $SO(q, y = z + 42; , \text{assert}(z > 42), B)$ evaluates to true. The function `new_edges` uses the predicate SO to get the operations with replaced variables based on the pattern ρ , in the previous example, it is $\text{assert}(z > 42)$. The function places the new operation after (A) or before (B) the edge (separated by location l^{new}) in the original CFA C. Lastly, the algorithm adds the new locations into C' in line 14.

$$SO(q^i, op^C, op^{A_i}, X) = \exists(q^i, \rho, \overline{op}^{A_i}, X, \cdot) \in \delta^i : \text{match}(\rho, op^C) \wedge op^{A_i} = \text{sub}(\overline{op}^{A_i}, \text{vars}(\rho, op^C)) \quad (3)$$

$$\begin{aligned} \text{new_edges}((l, q^i)) = \\ \bigcup_{\substack{(x, y, X) \in \\ \{(\mathcal{A}_i, C, B), \\ (C, \mathcal{A}_i, A)\}}} \{ (l, op^x, l^{new}), (l^{new}, op^y, l') \mid (l, op^C, l') \in G \wedge SO(q^i, op^C, op^{A_i}, X) \} \end{aligned} \quad (4)$$

Example 1. Consider the program from Fig. 1 and the property "If the execution of a program does at least one iteration of the loop, the value of x will always be at most 136". The property can be formalized as the instrumentation automaton in Fig. 3a. Applying the sequentialization operator to the IA in Fig. 3a and the CFA in Fig. 1 results in the CFA in Fig. 3b. The function `initialize_automata` returns the same IA as there is nothing to be instantiated in this example. It then initializes `waitlist` with the only pair (l_0, q_0) .

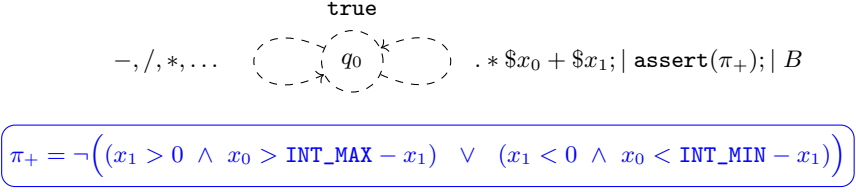


Fig. 5: Example instrumentation automaton for the *no-overflow* property

The most important explored pairs are (l_0, q_0) , (l_6, q_1) , and (l_6, q_2) , as they are the only pairs for which the annotation α holds. For example, let us focus on the CFA edge $(l_6, [x < 127], l_7)$ and the IA transition from q_1 to q_2 . The predicate $SO(q_1, [x < 127], looped = 1; A)$ is satisfied because the edge matches pattern **cond**, thus $\text{new_edges}((l_6, q_1)) = \{(l_6, [x < 127], l_6^{q_1}), (l_6^{q_1}, looped = 1; , l_7)\}$.

```
int main(void) {
  int looped = 0; //
  unsigned int x = 0;
  assert(x >= 0);
  int y = 1;

  while (x < 127) {
    looped = 1; //
    x = x + y;
    y = y + 1;
  }
  assert(x <= 136 || looped == 0); //
}
```

Fig. 4: Instrumented program; lines marked with *//* are inserted operations

Instrumentation Operator. The instrumentation operator \uplus uses the output CFA from the sequentialization operator and the original program as inputs. It iterates through every statement of the program and in case there are some newly inserted operations in the corresponding CFA location, it adds them before or after the operation in the original program based on their order in the input CFA.

Example 2. Applying the instrumentation operator \uplus to the CFA in Fig. 3b and the original program in Fig. 1 (left) results in the program in Fig. 4.

5 Specifications as Instrumentation Automata

To study the performance of reachability analyzers when applied to different specifications, we focus our experiments on the transformation of *no-overflow*, *termination*, and *memory cleanup* (as defined in SV-COMP [9]) to reachability. This section shows how to formalize the specifications as instrumentation automata.

No-Overflow. According to SV-COMP, the specification *no-overflow* is violated by a given program if there exists an execution of the program that contains an operation with a signed-integer value that does not fit into the type of the signed integer. The motivation for this specification is that programs should be well-defined, and an overflow of signed integers is an undefined behavior according to the C99 standard [60]. To simplify the presentation, let us assume that a program consists only of signed-integer variables. In our implementation, we track the types of the variables and the expressions on the edges of the CFA, and instrument only the operations with signed-integer results. As explained in Sect. 3

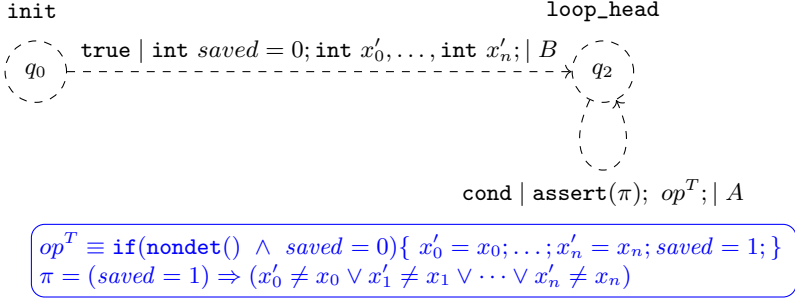


Fig. 6: Example instrumentation automaton for the *termination* property

complex arithmetic expressions are decomposed into multiple CFA edges, each containing one operation. Figure 5 shows the IA corresponding to the no-overflow specification. We display only the transition for addition since the transitions for the other operations are analogous. The automaton matches every operation and adds the corresponding condition as an **assert** before the operation. For example, before an addition, if x_1 is positive, then the result of $x_0 + x_1$ should not be larger than `INT_MAX`, and if x_1 is negative, then the result of $x_0 + x_1$ should not be smaller than `INT_MIN`. The operations together with the necessary conditions to prevent the overflows can be found listed online [61].

Termination. Figure 6 shows the instrumentation automaton for the *termination* property. It is based on a transformation of liveness properties to safety properties for finite systems [62]. To find an infinite execution, the instrumentation monitors the visited states of the execution. If a program state is encountered twice during the execution of a loop, a non-terminating execution has been found. As a preprocessing step, our implementation traverses the whole CFA and initializes an automaton for each loop. For every loop, it collects all variables that are in scope and initializes their shadow copies x'_0, \dots, x'_n . Each time the loop-head is visited, the instrumented program can make a non-deterministic choice to save the state if no state has been saved before. This is performed by the operation op^T . An assertion ensures that if the state was saved previously then the current state is different. A violation of the assertion means that the execution encountered the same state twice, and hence, it can repeat the loop infinitely often. Note that the transformation is complete but not sound, because dynamic structures like linked lists can have unbounded executions without visiting the same state twice.

Memory Cleanup. We assume in this transformation that all memory-allocation functions are called directly and not through a function pointer. Figure 7 shows an IA for this property. It non-deterministically decides to track the pointer being allocated and checks if the tracked pointer has been deallocated when the program terminates. The transition from q_0 to q_1 initializes the tracking of the pointer. In Fig. 7, we draw only one self-loop in q_1 but it actually represents four distinct edges, one for each of the lines above the arrow. The result of every memory-allocating function such as `malloc` and `calloc` is non-deterministically assigned to the tracking pointer `ptr`. For `realloc`, we first check if the pointer being

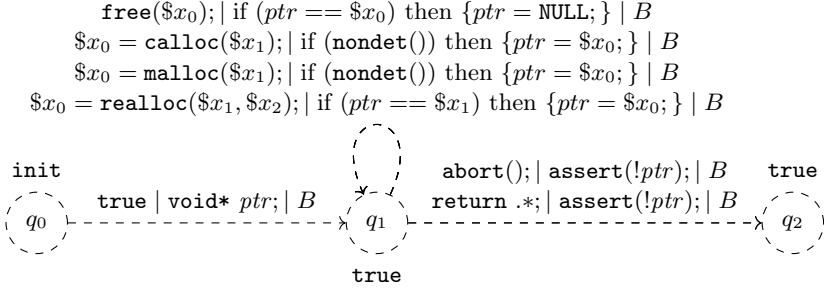


Fig. 7: Example instrumentation automaton for the *memory-cleanup* property

reallocated is the same as the one being tracked. If it is, we update the tracking pointer. When freeing memory by using `free`, we have to check if the pointer is the one currently being tracked. If it is, we set it to null, i.e., we are currently not tracking any pointer. Finally, when exiting the program, demonstrated through the state q_2 , we check if the tracking pointer is null. If not, then there exists at least one execution such that the pointer is not deallocated.

6 Evaluation

The evaluation of our approach addresses the following research questions:

RQ 1 (Modularity): Do transformations enable verifiers that only support reachability to verify properties for which they do not have integrated support?

RQ 2 (Effectiveness): Are state-of-the-art reachability verifiers combined with transformations *effective* compared to state-of-the-art verifiers with integrated specification support?

RQ 3 (Efficiency): Are state-of-the-art reachability verifiers combined with transformations *efficient* compared to state-of-the-art verifiers with integrated specification support?

RQ 4 (Degradation): Is there a performance difference between a verifier using a program transformation on the input program instead of during the analysis?

The proposed research questions divide the evaluation into three parts to provide answers for our initial motivation. First, RQ 1 shows that verifiers supporting only reachability can be adapted to verify other properties without additional engineering effort. Second, RQ 2 and RQ 3 evaluate if the fine-tuned reachability analyses of the best-performing verifiers can be as effective and efficient as state-of-the-art verifiers with integrated specification support. This supports a clear separation of concerns: Boosting the performance of the reachability analysis improves the performance on multiple specifications. Finally, RQ 4 evaluates whether encoding the transformation as a C program results in a performance loss compared to encoding it directly within the verification algorithm.

Transformations for no-overflow and termination do not put any additional requirements on the reachability analyzer. For memory cleanup, the verifiers need to handle memory allocation and deallocation correctly, even after the transformation. The verifiers used as reachability analyzers in RQ 1 do not participate in

Table 2: Verifiers used in the experiments

Verifier	Version	reachability	no-overflow	memory cleanup	termination
2LS [67]	[68]	✓	✗	✗	✓
CPACHECKER [69]	[70]	✓	✓	✓	✓
CPA-BAM-SMG [71]	[71]	✓	✓	✓	✓
CPV [72]	[73]	✓	✗	✗	✗
EMERGENTHETA [12]	[74]	✓	✗	✗	✗
PREDATORHP [75]	[76]	✓	✗	✓	✗
SYMBIOTIC [52]	[77]	✓	✓	✓	✓
THETA [13]	[78]	✓	✗	✗	✗
UAUTOMIZER [79]	[80]	✓	✓	✓	✓
UTAIPAN [81]	[82]	✓	✓	✗	✗

the memory-safety category of SV-COMP, hence, we assume they do not model memory with sufficient precision. Therefore, we answer only RQ 2 and RQ 3 for memory cleanup. We also exclude memory cleanup from RQ 4 since there is no internal transformation for memory cleanup inside CPACHECKER. We do not include the time for the transformation in the comparison as it is negligible.

Benchmark Set. To answer the proposed research questions, we use a subset with 2529 tasks for no-overflow, a subset with 779 tasks for termination, and the full set with 41 tasks⁴ for memory cleanup of the [SV-Benchmarks](#) collection at its SV-COMP 2025 version [63], the largest dataset of C programs with known verification verdicts for several properties. The chosen subsets of the benchmark set do not include programs containing dynamic data structures and arrays for property termination, and do not include programs with recursive function calls for property no-overflow. These programs were not included in the evaluation because our current implementation does not support these program features. If the expected verdict for a verification task is *true* (property holds), we call it a *proof*, if it is *false* (property does not hold), we call it an *alarm*.

Verifiers Evaluated. Table 2 lists all evaluated verifiers together with their supported properties. We selected sound⁵ and open-source⁶ verifiers that participated in SV-COMP 2024.⁷ We add *-R* to the name of a verifier when used to verify a transformed program (for reachability). For program transformation, we used version 1.0.1 of TRANSVER, which uses CPACHECKER at commit 66247485.

Benchmark Environment. For conducting our evaluation, we use BENCHEXEC to ensure reliable benchmarking [83]. 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 24.04 as operating system. Each verification task is limited to 900s of CPU time, 15 GB of memory, and 1 physical core

⁴ We excluded the tasks in `Juliet.set` because they were not used in SV-COMP 2024.

⁵ PROTON [64] won the termination category, but applies unsound approximations.

⁶ VERIABS_L [65] and VERIABS [66], 1st and 2nd place in the reachability category, are not open-source.

⁷ We used the 2025 version of CPV, because the 2024 version is based on cgroups v1.

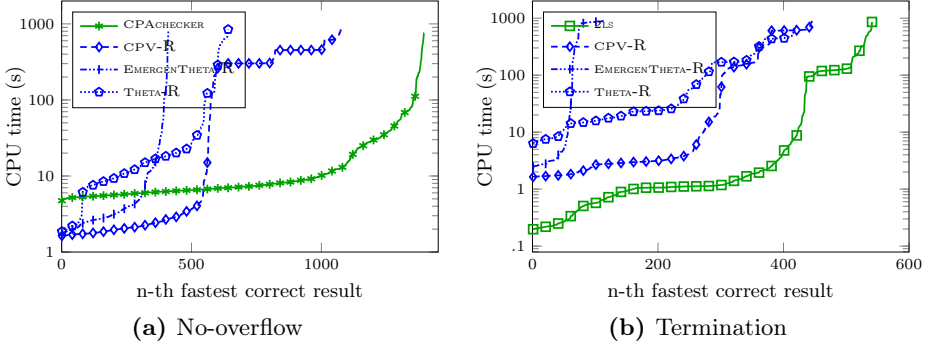


Fig. 8: *RQ 1 (Modularity)*: Quantile plots comparing reachability verifiers (without integrated support for the properties) on the transformed programs with the third-best verifier (with integrated support of the properties) in the respective SV-COMP 2024 category on the original programs

(2 processing units). The limits for time and memory are the same as used in SV-COMP 2024, but the competition used 2 physical cores (4 processing units).

6.1 RQ 1: Modularity

Since **TRANSVER** produces C programs, it directly allows any verifier for C programs that supports reachability to also analyze other specifications. We consider three verifiers that supported only reachability in SV-COMP 2024, namely CPV, THETA, and EMERGENTHETA. Figures 8a and 8b compare these verifiers with the third-best verifier of SV-COMP 2024 in the categories no-overflow and termination, respectively.

For no-overflow, CPV-R was only 317 tasks behind CPACHECKER, and for termination, CPV-R was only 94 tasks behind 2LS. The verifiers supporting only reachability could solve roughly two-thirds of the tasks on average that the third-best performing verifier could solve in the respective category. Notable is that CPV was 5th, THETA was 18th, and EMERGENTHETA was 20th for reachability in SV-COMP 2024 [9]. This indicates that reachability approaches can be useful for verifying different specifications even if they are outperformed on reachability tasks.

TRANSVER makes it possible for reachability verifiers to analyze specifications for which they do not have integrated support.

6.2 RQ 2: Effectiveness

To answer RQ 2 and RQ 3, we compare the results for two groups of verifiers: (a) the two best verifiers in the overall category in SV-COMP 2024, CPACHECKER and UAUTOMIZER, on the transformed programs, and (b) the best verifiers in SV-COMP 2024 in the respective category with integrated support for the property (cf. [9, Table 10], also Table 2) on the original tasks.

Table 3: *RQ 2 (Effectiveness)*: Results for 2529 transformed no-overflow tasks

Results (#Tasks)		CPACHECKER	UAUTOMIZER	UTAIPAN	CPACHECKER-R	UAUTOMIZER-R
Correct	2 529	1 396	1 852	1 856	1 583	1 731
Proofs	2 034	1 046	1 450	1 456	1 197	1 376
Alarms	495	350	402	400	386	355
Incorrect		1	0	0	2	1
Proofs		0	0	0	2	1
Alarms		1	0	0	0	0

Table 4: *RQ 2 (Effectiveness)*: Results for 779 transformed termination tasks

Results (#Tasks)		2LS	UAUTOMIZER	CPACHECKER-R	UAUTOMIZER-R
Correct	779	541	411	396	515
Proofs	384	224	268	118	294
Alarms	395	317	143	278	221

No-Overflow. Table 3 shows that UTAIPAN and UAUTOMIZER were able to provide 1856 (73 %) and 1852 (73 %) correct results, respectively. UAUTOMIZER-R solved 1731 (68 %) tasks correctly. CPACHECKER-R was able to find 151 more proofs and 36 more alarms than CPACHECKER with its integrated no-overflow analysis. We inspected all incorrect results and concluded that they were not caused by our transformation, since the other reachability analyzers were able to solve them correctly. It is expected that reachability and no-overflow algorithms are conceptually similar as they are both safety properties. However, the transformation allows us to use other algorithms for reachability which leads to an increase of the performance for CPACHECKER. In this particular case, the difference is likely due to the overflow analysis of CPACHECKER being based on predicate abstraction [84] and the reachability analysis being a portfolio approach including k-Induction, predicate analysis, and value analysis. With TRANSVER, we can leverage the strength of the portfolio for no-overflow tasks.

Termination. Table 4 shows the results for termination. UAUTOMIZER-R performs better than the termination analysis of UAUTOMIZER. It was able to solve 515 (66 %) of the tasks, and provide 26 more proofs and 78 more alarms than UAUTOMIZER with its integrated termination analysis. In contrast to no-overflow, where the performance gain could be attributed to the used algorithms, the difference in the performance for termination is more likely due to the conceptual differences between the verification approaches, because the algorithms developed to analyze termination are usually different from the algorithms for reachability. Thanks to the transformation framework, we had identified 17 tasks with undefined behavior in the form of signed-integer overflows. They were excluded in SV-COMP 2025.⁸

⁸ Since the termination behavior is not defined in this case, we removed them from the comparison and from SV-Benchmarks: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1543

Table 5: *RQ 2 (Effectiveness)*: Results for 41 transformed memory-cleanup tasks

Results (#Tasks)		PREDATORHP	SYMBIOTIC	CPA-BAM-SMG	CPACHECKER-R	UAUTOMIZER-R
Correct	41	35	39	34	33	23
proofs	2	1	2	1	0	0
alarms	39	34	37	33	33	23

Memory Cleanup. Table 5 shows the results of the comparison of the verifiers on the transformed memory-cleanup tasks. The results show no large discrepancy between the different verifiers. However, it is notable that CPACHECKER-R could solve almost as many tasks as CPA-BAM-SMG in this case and that it is close to the performance of PREDATORHP which specializes in memory analysis.

Threats to Validity. As mentioned in paragraph “Benchmark Set”, the benchmark set we used contains only a subset of the programs from SV-COMP, and hence, the evaluation on the full dataset could provide a different result.

The experiments show that the combination of the transformations and reachability verifier can outperform state-of-the-art verifiers for the other specifications. For all the properties, we can see that the best performing verifiers are verifiers with integrated support of the property.

6.3 RQ 3: Efficiency

To report the results on efficiency, we show quantile plots for the correct results of the three considered specifications in Fig. 9.

No-Overflow. Figure 9a shows that while CPACHECKER-R can solve more tasks than CPACHECKER, it uses a bit more CPU time overall. This can be explained by the configuration of CPACHECKER: the integrated no-overflow support is based on predicate analysis only, while, thanks to the flexibility gained by the transformation, we now use CPACHECKER’s sequential portfolio of reachability analyses for CPACHECKER-R.

Termination. Figure 9b shows that UAUTOMIZER-R on the transformed programs is more efficient than UAUTOMIZER on the original tasks, solving more tasks in less CPU time overall.

Memory Cleanup. The plot shows that there is no large difference in efficiency between the verifiers, besides about 10 seconds of JVM startup time for UAUTOMIZER and CPACHECKER. The efficiency is expected to be slightly worse on transformed programs, because the transformation introduces a non-deterministic choice for each allocation, making the programs more difficult to verify.

CPACHECKER-R tends to be less efficient compared to state-of-the-art verifiers with integrated support of the specifications. For UAUTOMIZER, there is a significant improvement of the efficiency for the termination specification.

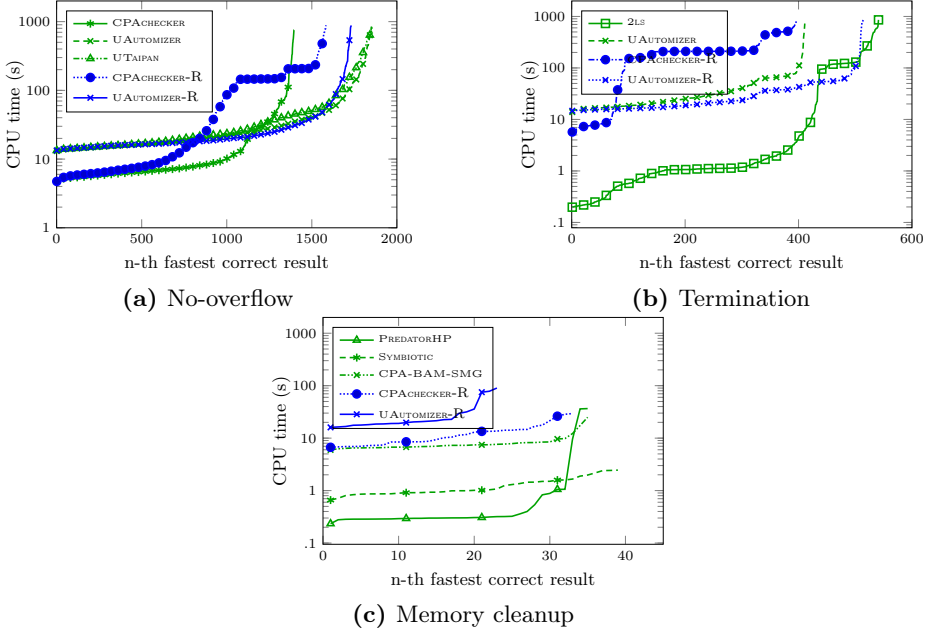


Fig. 9: *RQ 3 (Efficiency)*: Results for verifiers on original and transformed tasks

6.4 RQ 4: No Degradation

Some of the verifiers internally transform various properties to reachability. They usually do it by instrumenting their intermediate representation of an input program or by reflecting the assertion checks in their analyzing algorithm. There are two algorithms in CPACHECKER applying the latter. Figure 10 shows the CPU time in seconds for correctly solved tasks by both approaches, using the transformed programs versus using the internal transformation of the property.

For termination (+), both approaches used bounded model checking [85] as the reachability algorithm. It usually solves the task very quickly or does not solve it at all. We can see that for a lot of the tasks, there is a difference between 1-20 seconds if we represent the specification in the input program which is not too much in relation to the 900 seconds time limit. There were many cases where the reachability analysis was faster than the integrated analysis. This is probably due to a better-tuned BMC implementation for reachability.

For no-overflow (×), both approaches used predicate abstraction [84] as the reachability algorithm. The number of tasks solved by both approaches is much larger than for termination. For most of the tasks, there is no clear overhead in either direction, though there are a few outliers in both directions.

There is no clear degradation of efficiency using the external transformation for no-overflow. For termination, there is a slow-down in some cases, which is balanced by a speed-up in more other cases. In sum, the modular transformation outside the verifier does not lead to a general degradation of the performance.

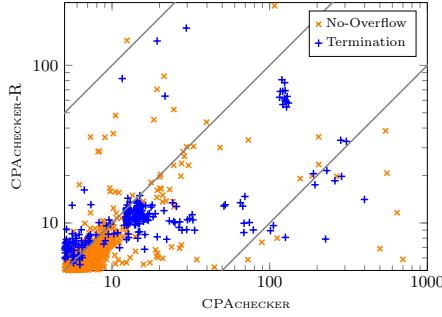


Fig. 10: *RQ 4 (No degradation)*: CPU time of program transformation using internal (CPACHECKER) vs. external (TRANSVER + CPACHECKER-R) transformation

7 Conclusion

Developing a verifier for software verification is challenging and requires a large engineering effort. The effort is even larger for supporting various specifications. Verification tools sometimes use internal transformations to mitigate the development time. However, these transformations are usually not modular and have to be developed and maintained separately for every verifier, and for each specification. Our contribution is to offer a new modular framework, implemented in TRANSVER, which separates the concerns of a high-performance reachability algorithm from supporting other specifications. We demonstrated how the framework works by implementing the transformations for three interesting specifications.

We showed that the construction of new verifiers as a composition of an off-the-shelf transformation and a following off-the-shelf reachability analysis is usually also efficient and effective, and can compete with (and sometimes outperform) state-of-the-art verifiers for no-overflow and termination analysis. Furthermore, the experiments demonstrate that our approach can enable verifiers like CPV or THETA to be competitive in the verification of properties for which they did not have integrated support so far. There was no significant general performance regression on transformed programs compared to using internal transformations.

Data-Availability Statement. A reproduction package containing all data and verifiers used for the experiments is available [86].

Funding Statement. This project was funded by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY) and the Free State of Bavaria.

References

1. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
2. Griggio, A., Jonáš, M.: KRATOS2: An SMT-based model checker for imperative programs. In: Proc. CAV. pp. 423–436. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_20
3. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software

- verification with CPACHECKER 3.0: Tutorial and user guide. In: Proc. FM. pp. 543–570. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_30
4. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST query language for software verification. In: Proc. SAS. pp. 2–18. LNCS 3148, Springer (2004). https://doi.org/10.1007/978-3-540-27864-1_2
 5. Ball, T., Rajamani, S.K.: SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, Microsoft Research (2002)
 6. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Proc. POPL. pp. 1–3. ACM (2002). <https://doi.org/10.1145/503272.503274>
 7. O. Šerý: Enhanced property specification and verification in BLAST. In: Proc. FASE. pp. 456–469. LNCS 5503, Springer (2009). https://doi.org/10.1007/978-3-642-00593-0_32
 8. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
 9. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
 10. Metta, R., Medicherla, R.K., Karmarkar, H.: VERIFUZZ: Fuzz centric test generation tool (competition contribution). In: Proc. FASE. pp. 341–346. LNCS 13241, Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_20
 11. Metta, R., Medicherla, R.K., Chakraborty, S.: BMC+FUZZ: Efficient and effective test generation. In: Proc. DATE. pp. 1419–1424. IEEE (2022). <https://doi.org/10.23919/DATE54114.2022.9774672>
 12. Bajczi, L., Szekeres, D., Mondok, M., Ádám, Z., Somorjai, M., Telbisz, C., Dobos-Kovács, M., Molnár, V.: EMERGENTHETA: Verification beyond abstraction refinement (competition contribution). In: Proc. TACAS (3). pp. 371–375. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_23
 13. Bajczi, L., Telbisz, C., Somorjai, M., Ádám, Z., Dobos-Kovács, M., Szekeres, D., Mondok, M., Molnár, V.: THETA: Abstraction based techniques for verifying concurrency (competition contribution). In: Proc. TACAS (3). pp. 412–417. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_30
 14. Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Xia, T., Zheng, X.: A modular program-transformation framework for reducing specifications to reachability. arXiv/CoRR **2501**(16310) (January 2025). <https://doi.org/10.48550/arXiv.2501.16310>
 15. Partsch, H., Steinbrüggen, R.: Program transformation systems. ACM Comput. Surv. **15**(3), 199–236 (1983). <https://doi.org/10.1145/356914.356917>
 16. Visser, E.: A survey of strategies in program-transformation systems. In: Proc. WRS. pp. 109–143. ENTCS 57, Elsevier (2001). [https://doi.org/10.1016/S1571-0661\(04\)00270-1](https://doi.org/10.1016/S1571-0661(04)00270-1)
 17. Beyer, D., Lee, N.Z.: The transformation game: Joining forces for verification. In: Principles of Verification: Cycling the Probabilistic Landscape. pp. 175–205. LNCS 15262, Springer (2024). https://doi.org/10.1007/978-3-031-75778-5_9
 18. Fischer, B., Inverso, O., Parlato, G.: CSEQ: A concurrency pre-processor for sequential C verification tools. In: Proc. ASE. pp. 710–713. IEEE (2013). <https://doi.org/10.1109/ASE.2013.6693139>
 19. Inverso, O., Nguyen, T.L., Fischer, B., La Torre, S., Parlato, G.: LAZY-CSEQ: A context-bounded model checking tool for multi-threaded C programs. In: Proc. ASE. pp. 807–812. IEEE (2015). <https://doi.org/10.1109/ASE.2015.108>

20. 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). https://doi.org/10.1007/3-540-45937-5_16
21. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
22. Alglave, J., Donaldson, A.F., Kröning, D., Tautschnig, M.: Making software verification tools really work. In: Proc. ATVA. pp. 28–42. LNCS 6996, Springer (2011). https://doi.org/10.1007/978-3-642-24372-1_3
23. Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M.: A unifying approach for control-flow-based loop abstraction. In: Proc. SEFM. pp. 3–19. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_1
24. Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M.: CEGAR-PT: A tool for abstraction by program transformation. In: Proc. ASE. pp. 2078–2081. IEEE (2023). <https://doi.org/10.1109/ASE56229.2023.00215>
25. 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>
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
27. 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
28. Madhukar, K., Wachter, B., Kröning, D., Lewis, M., Srivas, M.K.: Accelerating invariant generation. In: Proc. FMCAD. pp. 105–111. IEEE (2015). <https://doi.org/10.1109/FMCAD.2015.7542259>
29. Bodden, E., Hendren, L.: The CLARA framework for hybrid typestate analysis. *Softw. Tools Technol. Transf.* **14**(3), 307–326 (2012). <https://doi.org/10.1007/s10009-010-0183-5>
30. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: Checking memory safety with BLAST. In: Proc. FASE. pp. 2–18. LNCS 3442, Springer (2005). https://doi.org/10.1007/978-3-540-31984-9_2
31. Zhang, Y., Xie, X., Li, Y., Chen, S., Zhang, C., Li, X.: EndWatch: A practical method for detecting non-termination in real-world software. In: Proc. ASE. pp. 686–697 (2023). <https://doi.org/10.1109/ASE56229.2023.00061>
32. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.* **149**(1), 79–96 (2006). <https://doi.org/10.1016/j.entcs.2005.11.018>
33. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7
34. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Le Gall, P.: METACSL: Specification and verification of high-level properties. In: Proc. TACAS, Part 1. pp. 358–364. LNCS 11427, Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_22
35. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Gall, P.L.: Methodology for specification and verification of high-level requirements with METACSL. In: Proc. FormaliSE. pp. 54–67 (2021). <https://doi.org/10.1109/FormaliSE52586.2021.00012>
36. Blatter, L., Kosmatov, N., Le Gall, P., Prevosto, V.: RPP: Automatic proof of relational properties by self-composition. In: Proc. TACAS. pp. 391–397 (2017). https://doi.org/10.1007/978-3-662-54577-5_22

37. Blatter, L., Kosmatov, N., Le Gall, P., Prevosto, V., Petiot, G.: Static and dynamic verification of relational properties on self-composed C code. In: Proc. TAP. pp. 44–62 (2018). https://doi.org/10.1007/978-3-319-92994-1_3
38. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. *IEEE Trans. Softw. Eng.* **30**(1), 3–16 (2004). <https://doi.org/10.1109/TSE.2004.1265732>
39. Harman, M.: We need a testability transformation semantics. In: Proc. SEFM. pp. 3–17. LNCS 10886, Springer (2018). https://doi.org/10.1007/978-3-319-92970-5_1
40. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. *ACM Trans. Softw. Eng. Methodol.* **31**(4), 57:1–57:69 (2022). <https://doi.org/10.1145/3477579>
41. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. pp. 184–203. LNCS 14624, Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_11
42. Beyer, D., Spiessl, M.: METAVAL: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
43. Necula, G.C.: Proof-carrying code. In: Proc. POPL. pp. 106–119. ACM (1997). <https://doi.org/10.1145/263699.263712>
44. Julien, S.: E-ACSL: Executable ANSI/ISO C specification language (2022), available at <http://frama-c.com/download/e-acsl/e-acsl.pdf>
45. Necula, G.C., McPeak, S., Weimer, W.: CCURED: Type-safe retrofitting of legacy code. In: Proc. POPL. pp. 128–139. ACM (2002). <https://doi.org/10.1145/503272.503286>
46. Condit, J., Harren, M., McPeak, S., Necula, G.C., Weimer, W.: CCURED in the real world. In: Proc. PLDI. pp. 232–244. ACM (2003). <https://doi.org/10.1145/781131.781157>
47. Jakobsson, A., Kosmatov, N., Signoles, J.: Fast as a shadow, expressive as a tree: hybrid memory monitoring for C. In: Proc. SAC. pp. 1765–1772. ACM (2015). <https://doi.org/10.1145/2695664.2695815>
48. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: Proc. ISMM. pp. 47–58. ACM (2017). <https://doi.org/10.1145/3092255.3092269>
49. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proc. ECOOP. pp. 327–353. Springer (2001). https://doi.org/10.1007/3-540-45337-7_18
50. Mahrenholz, D., Spinczyk, O., Schröder-Preikschat, W.: Program instrumentation for debugging and monitoring with AspectC++. In: Proc. ISIRC. pp. 249–256 (2002). <https://doi.org/10.1109/ISORC.2002.1003713>
51. Vitovská, M., Chalupa, M., Strejček, J.: SBT-Instrumentation: A tool for configurable instrumentation of LLVM bytecode. *arXiv/CoRR* **1810**(12617) (2018). <https://doi.org/10.48550/arXiv.1810.12617>
52. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Trtík, M., Zaoral, L., Ayaziová, P., Strejček, J.: SYMBIOTIC 10: Lazy memory initialization and compact symbolic execution (competition contribution). In: Proc. TACAS (3). pp. 406–411. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_29
53. 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). https://doi.org/10.1007/978-3-642-32469-7_14

54. Menezes, R., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M.R., Tihanyi, N., Korovin, K., Cordeiro, L.: ESBMC v7.4: Harnessing the power of intervals (competition contribution). In: Proc. TACAS (3). pp. 376–380. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_24
55. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: Proc. ASE. pp. 888–891. ACM (2018). <https://doi.org/10.1145/3238147.3240481>
56. Amilon, J., Esen, Z., Gurov, D., Lidström, C., Rümmer, P.: Automatic program instrumentation for automatic verification. In: Proc. CAV. pp. 281–304 (2023). https://doi.org/10.1007/978-3-031-37709-9_14
57. 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
58. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Handbook of Model Checking, pp. 27–73. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_2
59. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS (3). pp. 151–186. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9
60. American National Standards Institute: ANSI/ISO/IEC 9899-1999: Programming Languages — C. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA (1999)
61. INT32-C. Ensure that operations on signed integers do not result in overflow. <https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow>, [Accessed 2025-07-13]
62. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: Proc. FMICS. pp. 160–177. No. 2 in ENTSC 66, Elsevier (2002). [https://doi.org/10.1016/S1571-0661\(04\)80410-9](https://doi.org/10.1016/S1571-0661(04)80410-9)
63. Beyer, D., Strejček, J.: SV-Benchmarks: Benchmark set for software verification (SV-COMP 2025). Zenodo (2025). <https://doi.org/10.5281/zenodo.15012096>
64. Metta, R., Karmarkar, H., Madhukar, K., Venkatesh, R., Chakraborty, S.: PROTON: Probes for non-termination and termination (competition contribution). In: Proc. TACAS (3). pp. 393–398. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_27
65. Darke, P., Chimdyalwar, B., Agrawal, S., Venkatesh, R., Chakraborty, S., Kumar, S.: VERIABS: Scalable verification by abstraction and strategy prediction (competition contribution). In: Proc. TACAS (2). pp. 588–593. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_41
66. 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). <https://doi.org/10.1109/ASE.2019.00121>
67. Malík, V., Schrammel, P., Vojnar, T., Nečas, F.: 2LS: Arrays and loop unwinding (competition contribution). In: Proc. TACAS (2). pp. 529–534. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_31
68. Viktor, M., František, N., Schrammel, P., Vojnar, T.: 2LS. Zenodo (2023). <https://doi.org/10.5281/zenodo.10184626>
69. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPACHECKER 2.3

- with strategy selection (competition contribution). In: Proc. TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
70. Beyer, D., Wendler, P.: CPACHECKER release 2.3 (unix). Zenodo (2023). <https://doi.org/10.5281/zenodo.10203297>
 71. Vasilyev, A.: CPA-BAM-SMG (SV-COMP 2023). Zenodo (2022). <https://doi.org/10.5281/zenodo.10396261>
 72. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier (competition contribution). In: Proc. TACAS (3). pp. 365–370. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22
 73. Chien, P.C., Lee, N.Z.: CPV release 0.6. Zenodo (2024). <https://doi.org/10.5281/zenodo.14203582>
 74. Bajczi, L., Szekeres, D., Mondok, M., Molnár, V.: EMERGENTHETA - SV-COMP’24 verifier archive. Zenodo (2023). <https://doi.org/10.5281/zenodo.10198872>
 75. Peringer, P., Šoková, V., Vojnar, T.: PREDATORHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Proc. TACAS (2). pp. 408–412. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_30
 76. Šoková, V., Peringer, P., Vojnar, T., Kinšt, O.: PREDATORHP. Zenodo (2023). <https://doi.org/10.5281/zenodo.10183805>
 77. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Shandilya, S., Trtík, M., Zaoral, L., Strejček, J.: SYMBIOTIC 10: Submission to SV-COMP 2024. Zenodo (2023). <https://doi.org/10.5281/zenodo.10202594>
 78. Bajczi, L., Telbisz, C., Somorjai, M., Ádám, Z., Dobos-Kovács, M., Szekeres, D., Molnár, V.: THETA - SV-COMP’24 verifier archive. Zenodo (2023). <https://doi.org/10.5281/zenodo.10202679>
 79. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: ULTIMATE AUTOMIZER and the abstraction of bitwise operations (competition contribution). In: Proc. TACAS (3). pp. 418–423. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
 80. Dietsch, D., Bentele, M., Heizmann, M., Klumpp, D., Podelski, A., Schüssele, F.: ULTIMATE AUTOMIZER SV-COMP 2024. Zenodo (2023). <https://doi.org/10.5281/zenodo.10203545>
 81. Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: ULTIMATE TAIPAN 2023 (competition contribution). In: Proc. TACAS (2). pp. 582–587. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_40
 82. Dietsch, D., Bentele, M., Heizmann, M., Klumpp, D., Podelski, A., Schüssele, F.: ULTIMATE TAIPAN SV-COMP 2024. Zenodo (2023). <https://doi.org/10.5281/zenodo.10203547>
 83. 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>
 84. Beyer, D., Dangel, 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>
 85. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58**, 117–148 (2003). [https://doi.org/10.1016/S0065-2458\(03\)58003-2](https://doi.org/10.1016/S0065-2458(03)58003-2)
 86. Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Xia, T., Zheng, X.: Reproduction package for SPIN2025 article ‘TRANSVER: A modular program-transformation framework for reduction to reachability’. Zenodo (2025). <https://doi.org/10.5281/zenodo.15833440>