Cooperation Between Automatic and Interactive Software Verifiers

Dirk Beyer, Martin Spiessl, and Sven Umbricht

LMU Munich, Munich, Germany

Abstract. The verification community develops two kinds of verification tools: automatic verifiers and interactive verifiers. There are many such verifiers available, and there is steady progress in research. However, cooperation between the two kinds of verifiers was not yet addressed in a modular way. Yet, it is imperative for the community to leverage all possibilities, because our society heavily depends on software systems that work correctly. This paper contributes tools and a modular design to address the open problem of insufficient support for cooperation between verification tools. We identify invariants as information that needs to be exchanged in cooperation, and we support translation between two 'containers' for invariants: program annotations and correctness witnesses. Using our new building blocks, invariants computed by automatic verifiers can be given to interactive verifiers as annotations in the program, and annotations from the user or interactive verifier can be given to automatic verifiers, in order to help the approaches mutually to solve the verification problem. The modular framework, and the design choice to work with readily-available components in off-the-shelf manner, opens up many opportunities to combine new tools from existing components. Our experiments on a large set of programs show that our constructions work, that is, we constructed tool combinations that can solve verification tasks that the verifiers could not solve before.

Keywords: Software verification, Program analysis, Invariant generation, Automatic verification, Interactive verification, CPAchecker, Frama-C

1 Introduction

Software verification becomes more and more important, and large IT companies are investing into this technology [5,25,29]. There was a lot of progress in the past two decades and many software-verification tools exist [7,8,15,34,42]. But there are also obstacles that hinder the application of new technology in practice [3,35]. The verification tools can roughly be divided into two different flavors: automatic verifiers, which are more suited for automatic settings such as continuous-integration checks, and interactive verifiers, which can be fed with proof hints to solve verification tasks. These different tools have different strengths and often one verifier alone is not able to prove the correctness. Yet, the

B.-H. Schlingloff and M. Chai (Eds.): SEFM 2022, LNCS 13550, pp. 111–128, 2022.
https://doi.org/10.1007/978-3-031-17108-6_7

potential from cooperation between different kinds of verifiers is a largely unused technology, although it is expected to significantly improve the state of the art.

In this paper, we contribute ideas to bridge the gap between automatic and interactive verifiers by introducing cooperation between tools of both kinds. As a starting point, we identify invariants as the objects that we need to exchange. Then we investigate which interfaces are supported by different verification tools. As a result, we choose verification witnesses [12] and annotations [6] as containers for the invariants. We implement various transformers for exchanging invariants between the different interfaces. This results in a modular composition framework that is based on off-the-shelf components (in binary format). We can use existing components because we base our work on existing interfaces (witnesses and annotations).

Automatic verifiers, such as CBMC [28], CPACHECKER [18], GOBLINT [49], KORN [32], PESCO [48], SYMBIOTIC [26], ULTIMATE AUTOMIZER [39], and VERIABS [1] (alphabetic order, just to name a few, for a larger list we refer to a competition report [8]), usually take as input a program and a specification (a.k.a. verification task) and compute invariants, in order to prove correctness. The above-mentioned verifiers can save the computed invariants into a standard witness file for later use (e.g., for result validation).

Interactive verifiers, such as DAFNY [46], FRAMA-C [30], KEY [2], KIV [33], and VERIFAST [43] (alphabetic order, just to name a few, for a larger list we refer to a competition report [34]), usually take as input a program with an inlined specification (contracts, asserts), and during the verification process, the verification engineer can interact with the verifier by providing invariants and other information as annotations in the program.

The automatic verifiers use a standardized exchange format for verification witnesses [12], and thus, we can easily plug-in all of them. The interactive verifiers come each with their own annotation language. We decided to consider only ACSL [6], which is supported by FRAMA-C [30], as a starting point for our study, because it is well documented. In practice, many of these annotation languages are similar, so our results apply to other annotation languages as well.

Contributions. This paper contributes the following in order to enable new verification technology:

- We develop a novel compositional design to construct new tools for software verification from existing 'off-the-shelf' components:
  1. We construct interactive verifiers from automatic verifiers and validators.
  2. We construct result validators from interactive verifiers.
  3. We improve interactive verifiers by feeding them with invariants computed by automatic verifiers.
- We identified an appropriate benchmark set of verification tasks with verification witnesses that contain provably useful invariants. We also created second benchmark set with manually added ACSL annotations containing (inductive) loop invariants and assertions. In order to make our evaluation reproducible and to offer the invariants to other researchers for further experiments, we make both benchmark sets available.

- We make all components and transformations available as open source, such that other researchers and practitioners can reuse and experiment with them, and verify our results (see Sect. 5 for the data-availability statement).
- We perform a sound experimental evaluation on a large benchmark set to investigate the effectivity of the new compositions. The results are promising and suggest that such compositions are worth to be considered in practice.

Combinations like the proposed cooperation approach can significantly impact the way in which verification tools are used in practice. Currently, engineers need to use both kinds of verifiers, automatic and interactive, in isolation, but our study has shown that there is much potential in leveraging cooperation.

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

Transform Programs. This is not the first work to convert the semantics of witness validation into a program. Some existing approaches [14] focus on violation witnesses, while we solely focus on correctness witnesses. Most similar in this regard is MetaVal [21]. The main difference is that we preserve the program structure while MetaVal does an automaton product between the *control-flow automaton (CFA)* of the program and witness automaton, and turns the result back into a C program, which will result in a different syntactic structure.

Interact via Conditions. The approach *conditional model checking* [16] also achieves cooperation between verifiers, but is limited to automatic verifiers that support the condition format and the verifier that comes second uses the condition to restrict the part of the state space that is explored. Our framework supports more tools via the usage of standardized exchange formats, also considers interactive verifiers, and the second verifier still performs a full proof. Another approach that builds on conditions is *alternating conditional analysis* [36,37]. Here, the witness format is also used as standardized exchange format and multiple verifiers are supported. However, the focus is on violation witnesses whereas we are focussing on correctness witnesses. Instead of removing parts of the state space, we actually extend the property that needs to be checked, such that it is (potentially) easier to be proven. The same holds if we compare our component Witness2Assert to *reducer-based conditional model checking* [17]. While both approaches encode the important information into the original program, we actually would need to assume the invariants instead of asserting them in order to act as a reducer. Conditions are also used to improve testing [19,27,31].

Store and Exchange Proofs. Another parallel can be drawn to *proof-carrying code* [44,45,47], where the proof of correctness is stored alongside the program. We do the same here in cases where the added annotations actually suffice for a full proof by Frama-C, but we also have the possibility to generate partial proofs. Correctness witnesses are used to store intermediate results and to validate results [11]. Proofs are also stored in the area of theorem provers [38] (https://www.isa-afp.org/) and SAT solvers [40,41].

```
1
2  int main() {
3    unsigned int x = 0;
4    unsigned int y = 0;
5
6    while (nondet_int()) {
7      x++;
8
9      y++;
10   }
11   assert(x==y);
12   return 0;
13 }
```

Fig. 1. Example program with loop invariant `x==y`

```
1  //@ensures \return==0;
2  int main() {
3    unsigned int x = 0;
4    unsigned int y = 0;
5    //@loop invariant x==y;
6    while (nondet_int()) {
7      x++;
8      //@assert x==y+1;
9      y++;
10   }
11   assert(x==y);
12   return 0;
13 }
```

Fig. 2. Example program with ACSL annotations

2 Preliminaries

For our framework that enables cooperation between automatic and interactive verifiers we need to take into account the interfaces that each of them provide, i.e., how the information important for the verification process is communicated. For automatic verifiers there exists a common exchange format [12] in which verifiers export the program invariants they found. For interactive verifiers, we look at ACSL [6], the specification language that is e.g. used by Frama-C. In the following, we will quickly introduce these formats and the general verification problem we are looking at using a small example program that is depicted in Fig. 1.

For the rest of the paper, we will focus on reachability properties, though our approach can also be extended to work for other properties as well.[1] The crucial part of verifying reachability properties is to find the right loop invariants. In the example program this would be the fact that `x==y` always holds before each loop iteration. Please note that while this invariant is also present in the assertion in line 11, for more complicated programs it is generally not the case that we can find the invariants written in the code. Also, since there might be more than one loop in a program, a verifier might only partially succeed and therefore only be able to provide invariants for some of these loops, or only invariants that are not yet strong enough to prove the program correct. This is why cooperation by exchange of these discovered invariants can potentially lead to better results.

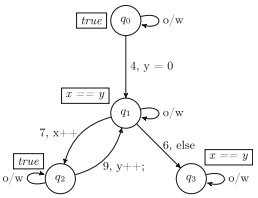
2.1 Verification Witnesses

In case an automatic verifier can prove our example program correct, information like a discovered invariant is normally made available as shown in Fig. 3a in the standard witness exchange format (described in [12], maintained at https://github.com/sosy-lab/sv-witnesses) as correctness witness. There are also

[1] Also, we will concentrate only on intraprocedural analysis, though our approach works for interprocedural analysis as well.

```
1  ...
2  <node id="q1">
3  <data key="invariant">( y == x )</data>
4  <data key="invariant.scope">main</data>
5  </node>
6  <edge source="q0" target="q1">
7  <data key="enterLoopHead">true</data>
8  <data key="startline">6</data>
9  <data key="endline">6</data>
10 <data key="startoffset">157</data>
11 <data key="endoffset">165</data>
12 </edge>
13 ...
```

(a) Encoding of an invariant in a GraphML-based correctness witness



(b) Example witness automaton for the program from Fig. 1

Fig. 3. Example of the witness format and automaton; o/w stands for otherwise, i.e., all other possible program transitions

violation witnesses in case a violation has been found, but since we are mainly interested in the invariants, we will focus on correctness witnesses and omit the prefix "correctness" for the rest of the paper.

Such a witness contains a graph representation of an observer automaton. Invariants can be given for nodes if they always hold when the witness automaton is in the corresponding state. The semantics of the witness is given by constructing the product of the witness automaton and the CFA of the program. This might lead to edge cases where the exact semantics depends on how the tool interpreting the witness constructs a CFA from the program, but in practice a witness can be written such that it is mostly robust against those differences. For further details on the semantics of the witness automata we refer the reader the existing literature [12].

There are currently some restrictions on the contents of an invariant: An invariant has to be a valid C expression that can be evaluated to an `int` at the current scope in the program. It may contain conjunctions and disjunctions but no function calls.

2.2 ACSL

Interactive verifiers rely on the user to provide the (non-trivial) invariants for the proof. An example can bee seen in Fig. 2, where the loop invariant has been added as ACSL annotation in line 5. Only when this information is externally

provided (usually by the user), an interactive verifier like Frama-C is able to prove that the assertion in line 11 can never be violated.

Loop annotations are only one of many kinds of annotation in ACSL. For example we can see a function contract in line 1 and an assertion in line 8. These annotations usually represent specifications which the implementation should adhere to, but they can also be seen as invariants, since they should hold for every possible program execution.

The basic building blocks of ACSL annotations are *logic expressions* that represent the concrete properties of the specification, e.g., `a + b > 0` or `x && y == z`. Logic expressions can be subdivided into terms and predicates, which behave similarly as terms and formulas in first-order logic. Basically, logic expressions that evaluate to a boolean value are predicates, while all other logic expressions are terms. The above example `a + b > 0` is therefore a predicate, while `a + b` is a term. We currently support only logic expressions that can also be expressed as C expressions, as they may not be used in a witness otherwise. Finding ways to represent more ACSL features is a topic of ongoing research.

ACSL also features different types of annotations. In this paper we will only present translations for the most common type of annotations, namely function contracts, and the simplest type, namely assertions. Our implementation also supports statement contracts and loop annotations.

All types of ACSL annotations when placed in a C source file must be given in comments starting with an `@` sign, i.e., must be in the form `//@ annotation` or `/*@ annotation */`. ACSL assertions can be placed anywhere in a program where a statement would be allowed, start with the keyword `assert` and contain a predicate that needs to hold at the location where the assertion is placed.

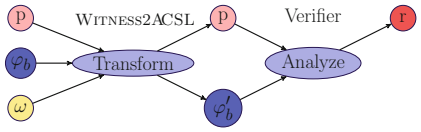
3 A Component Framework for Cooperative Verification

The framework we developed consists of three core components that allow us to improve interaction between the existing tools.

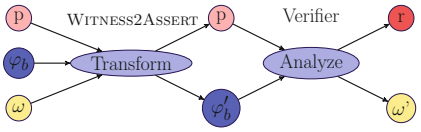
Witness2ACSL acts as transformer that converts a program and a correctness witness given as witness automaton where invariants are annotated to certain nodes, into a program with ACSL annotations.

ACSL2Witness takes a program that contains ACSL annotations, encodes them as invariants into a witness automaton and produces a correctness witness in the standardized GraphML format.

Witness2Assert is mostly identical to Witness2ACSL. The main difference is that instead of adding assertions as ACSL annotations to the program, it actually encodes the semantics of the annotations directly into the program such that automatic verifiers will understand them as additional properties to prove. On the one hand, this component enables us to check the validity of the ACSL annotations for which ACSL2Witness generated a witness, with tools that do not understand the annotation language ACSL. On the other hand, this component is also useful on its own, since it allows us to validate correctness witnesses and give

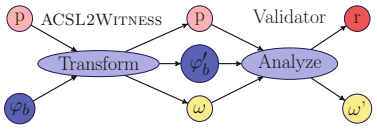


(a) Validator from Interactive Verifier(Frama-C) using Witness2ACSL

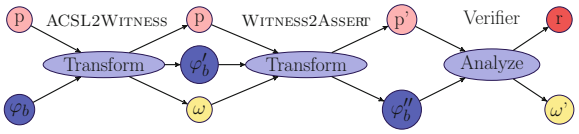


(b) Validator from Automatic Verifier using Witness2Assert

(c) Visualization of information in a witness using Witness2ACSL



(d) Interactive Verifier from Validator



(e) Interactive Verifier from Automatic Verifier

Fig. 4. Graphical visualization of the developed components to improve cooperation; we use the notation introduced in previous work [24]: p represents a program, ϕ_b a behavior specification, ω a witness, and r a verification result

witness producers a better feedback on how their invariants are interpreted and whether they are useful (validator developers can inspect the produced program).

These three components now enable us to achieve cooperation in many different ways. We can utilize a proposed component framework [24] to visualize this as shown in Fig. 4. The use case shown in Fig. 4a is to use Frama-C as a correctness witness validator. This is interesting because it can further reduce the technology bias (the currently available validators are based on automatic verifiers [4,11,13,21], test execution [14], and interpretation [50]). By using Witness2Assert instead of Witness2ACSL as shown in Fig. 4b we can also configure new correctness witness validators that are based on automatic verifiers, similar to what metaval [21] does, only with a different transformer. Figure 4c illustrates the use of Witness2ACSL (or similarly for Witness2Assert) to inspect the information from the witness as annotations in the program code.

The compositional framework makes it possible to leverage existing correctness witness validators and turn them into interactive verifiers that can understand ACSL, as shown in Fig. 4d. Since we also have the possibility now to construct a validator from an automatic verifier (Fig. 4b) we can turn automatic verifiers into interactive ones as depicted in Fig. 4e. While automatic verifiers can already make use of assertions that are manually added to the program, this now also allows us to use other types of high-level annotations like function contracts without having to change the original program.

3.1 Witness2ACSL

To create an ACSL annotated program from the source code and a correctness witness, we first need to extract *location invariants* from the witness, i.e., invariants that always hold at a certain program location (with program locations we refer to the nodes of the CFA here). We can represent location invariants as a tuple (l, ϕ) consisting of a program location l and an invariant ϕ. In general there is no one-to-one mapping between the invariants in the witness and this set of location invariants, since there might be multiple states with different invariants in the witness automaton that are paired with the same program location in the product with the CFA of the program. For extracting the set of location invariants, we calculate this product and then take the disjunctions of all invariants that might hold at each respective location.

3.2 ACSL2Witness

In order to convert the ACSL annotations present in a given program, we transform each annotation into a set of ACSL predicates that capture the semantics of those annotations and use the predicates as invariants in a witness. This mode of operation is based on two observations: Firstly, for a given ACSL annotation it is usually possible to find a number of ACSL assertions that are semantically equivalent to that annotation. For example, a loop invariant can be replaced by asserting that the invariant holds at the loop entry, i.e., before each loop iteration. Secondly, most ACSL assertions are logically equivalent to a valid invariant and can therefore be used in a witness. As mentioned in Sect. 2.2, we currently only support those predicates which can be converted into C expressions, which is a limitation of the witness format and might be lifted in future versions of the format.

3.3 Witness2Assert

This component is very similar to Witness2ACSL. The main difference is that instead of generating ACSL annotations we generate actual C code that encodes the invariants as assertions (i.e., additional reachability properties). This translation is sound since assertions added this way do not hide violations, i.e., every feasible trace that violates the original reachability property in the program before the modification will either still exist or have a corresponding trace that violates the additional reachability properties of the modified program. It is worth mentioning that this is an improvement compared to existing transformations like the one used in MetaVal [21], where the program is resynthesized from the reachability graph and the soundness can therefore easily be broken by a bug in MetaVal's transformation process.

4 Evaluation

We implemented the components mentioned in Sect. 3 in the software-verification framework CPAchecker. In our evaluation, we attempt to answer the following research questions:

- **RQ 1:** Can we construct interactive verifiers from automatic verifiers, and can they be useful in terms of effectiveness?
- **RQ 2:** Can we improve the results of, or partially automate, interactive verifiers by annotating invariants that were computed by automatic verifiers?
- **RQ 3:** Can we construct result validators from interactive verifiers?
- **RQ 4:** Are verifiers ready for cooperation, that is, do they produce invariants that help other verifiers to increase their effectiveness?

4.1 Experimental Setup

Our benchmarks are executed on machines running Ubuntu 20.04. Each of these machines has an Intel E5-1230 processor with 4 cores, 8 processing units, and 33 GB of RAM. For reliable measurements we use BenchExec [20]. For the automatic verifiers, we use the available tools that participated in the ReachSafety category of the 2022 competition on software verification (SV-COMP) in their submission version[2]. Frama-C will be executed via Frama-C-SV [22], a wrapper that enables Frama-C to understand reachability property and special functions used in SV-COMP. Unless otherwise noted we will use the EVA plugin of Frama-C. We limit each execution to 900 s of CPU time, 15 GB of RAM, and 8 processing units, which is identical to the resource limitations used in SV-COMP.

4.2 Benchmark Set with Useful Witnesses

In order to provide meaningful results, we need to assemble an appropriate benchmark set consisting of witnesses that indeed contain useful information, i.e., information that potentially improves the results of another tool.

As a starting point, we consider correctness witnesses from the final runs of SV-COMP 2022 [8,10]. This means that for one verification task we might get multiple correctness witnesses (from different participating verifiers), while for others we might even get none because no verifier was able to come up with a proof. We select the witnesses for tasks in the subcategory ReachSafety-Loops, because this subcategory is focussed on verifying programs with challenging loop invariants. This selection leaves us with 6242 correctness witnesses (without knowing which of those actually contain useful information).

For each of the selected witnesses we converted the contained invariants into both ACSL annotations (for verification with Frama-C) and assertions (for verification with automatic verifiers from SV-COMP 2022). Here we can immediately drop those witnesses that do not result in any annotations being generated, which results in 1931 witnesses belonging to 640 different verification tasks.

[2] https://gitlab.com/sosy-lab/sv-comp/archives-2022/-/tree/svcomp22/2022

Table 1. Impact of cooperation: in each row, a 'consuming' verifier is fed with information from witnesses of our benchmark set; 'Baseline' reports the number of programs that the verifier proved correct without any help; 'Improved via coop.' reports the number of programs that the verifier can prove *in addition*, if the information from the witness is provided

Consuming verifier	Benchmark tasks (434 total)		Projection on programs (230 total)	
	Baseline	Improved via coop.	Baseline	Improved via coop.
2LS	157	179	83	111
UAUTOMIZER	360	47	186	31
CBMC	281	53	142	28
CPACHECKER	300	69	149	53
DARTAGNAN	280	82	139	51
ESBMC	239	133	121	76
GAZER-THETA	266	118	135	64
GOBLINT	38	106	21	47
UKOJAK	191	134	97	76
KORN	183	46	98	27
PESCO	180	162	87	99
PINAKA	258	105	127	59
SYMBIOTIC	349	51	174	32
UTAIPAN	334	65	172	37
VERIABS	343	31	186	28
FRAMA-C	211	31	105	20

We then run each verifier for each program where annotations have been generated, once with the original, unmodified program, and n times with the transformed program for each of the n witnesses. This allows us determine whether any improvement was achieved, by looking at the differences between verification of the unmodified program versus verification of a program that has been enhanced by information generated from some potentially different tool. Using this process, we further reduce our benchmark set of witnesses to those that are useful for at least one of the verifiers and thus enable cooperation. This leads to the final set of 434 witnesses that evidently contain information that enables cooperation between verifiers. These witnesses correspond to 230 different programs from the SV-Benchmarks repository (https://github.com/sosy-lab/sv-benchmarks). We made this benchmark set available to the community in a supplementary artifact of this paper [23].

4.3 Experimental Results

RQ 1. For the first research question, we need to show that we can construct interactive verifiers from automatic verifiers, and that they can be useful in terms of effectiveness. By "interactive verifier", we mean a verifier that can verify

more programs correct if we feed it with invariants, for example, by annotating the input program with ACSL annotations. Using our building blocks from Sect. 3, an interactive verifier can be composed as illustrated in Fig. 4e (that is, configurations of the form ACSL2Witness|Witness2Assert|Verifier). For a meaningful evaluation we need a large number of annotated programs, which we would be able to get if we converted the witnesses from SV-COMP using Witness2ACSL in advance. But since the first component ACSL2Witness in Fig. 4e essentially does the inverse operation, we can generalize and directly consider witnesses as input, as illustrated in Fig. 4b (that is, configurations of the form Witness2Assert|Verifier).

Now we look at the results in Table 1: The first row reports that cooperation improves the verifier 2ls in 179 cases, that is, there are 179 witnesses that contain information that helps 2ls to prove a program that it could not prove without the information. In other words, for 179 witnesses, we ran Witness2Assert to transform the original program to one in which the invariants from the witness were written as assertions, and 2ls was then able to verify the program. Since there are often several witnesses for the same program, 2ls verified in total 111 unique unique programs that it was not able to verify without the annotated invariants as assertion.

In sum, the table reports that many programs that could not be proved by verifiers when ran on the unmodified program, could be proved when the verifier was given the program with invariants. Since we were able to show the effect using generated witnesses, it is clear that manually provided invariants will also help the automatic verifiers to prove the program. We will continue this argument in Sect. 4.4.

RQ 2. For the second research question, we need to show that our new design can improve the results of interactive verifiers by annotating invariants that were computed by automatic verifiers. Using our building blocks from Sect. 3, we assemble a construction as illustrated in Fig. 4a (i.e., configurations of the form Witness2ACSL|Verifier). We take a program and a witness and transform the program to a new program that contains the invariants from the witness as ACSL annotations.

Let us consider the last row in Table 1: Frama-C is able to prove 20 programs correct using invariants from 31 witnesses. Those 31 witnesses were computed by automatic verifiers, and thus, we can conclude that our new design enables using results of automatic verifiers to help the verification process of an interactive verifier.

RQ 3. For the third research question, we need to show that we can construct result validators from interactive verifiers and that they can effectively complement existing validators. A results validator is a tool that takes as input a verification task, a verdict, and a witness, and confirms or rejects the result. In essence, due to the modular components, the answer to this research question can be given by the same setup as for RQ 2: If the interactive verifier (Frama-C) was able to prove the program correct, then it also has proved that the invariants provided by the witnesses were correct, and thus, the witness should be confirmed. Frama-C has confirmed 31 correctness witnesses.

Table 2. Proof of cooperation: for each 'producing' verifier, we report the number of correctness witnesses that help another verifier to prove a program which it otherwise could not; we also list the number of cases where this cooperation was observed (some witnesses improve the results of multiple verifiers); we omit producers without improved results

Producing verifier	Useful witnesses	Cases of cooperation
2LS	1	1
CBMC	20	22
CPACHECKER	148	533
GOBLINT	2	3
GRAVES-CPA	151	823
KORN	10	15
PESCO	78	271
SYMBIOTIC	5	10
UAUTOMIZER	19	70
Sum	434	1748

New validators that are based on a different technology are a welcome complement because this reduces the technology bias and increases trust. Also, the proof goals for annotated programs might be interesting for verification engineers to look at, even or especially when the validation does not succeed completely.

RQ 4. For the fourth research question, we report on the status of cooperation-readiness of verifiers. In other words, the question is if the verifiers produce invariants that help other verifiers to increase their effectiveness.

In Table 2 we list how many useful witnesses each verifier contributed to our benchmark set of useful witnesses. The results show that there are several verifiers that produce significant amounts of witnesses that contain invariants that help to improve results of other verifiers.

4.4 Case Study on Interactive Verification with Manual Annotations

So far, we tested our approach using information from only the SV-COMP witnesses. For constructing interactive verifiers, we would also like to evaluate whether our approach is useful if the information is provided by an actual human in the form of ACSL annotations.

ACSL Benchmark Set. To achieve this, we need a benchmark set with tasks that contain sufficient ACSL annotations and also adhere to the conventions of SV-COMP. Since to our knowledge such a benchmark set does not exist yet, we decided to manually annotate assertions and loop invariants to the tasks from the SV-Benchmarks collection ourselves. While annotating all of the benchmark tasks is out of scope, we managed to add ACSL annotations to 125 tasks from the ReachSafety-Loops subcategory. This subcategory is particularly relevant, since it contains a selection of programs with interesting loop invariants. The loop invariants we added are sufficient to proof the tasks correct in a pen-and-paper,

Table 3. Case study with 125 correct verification tasks where sufficient, inductive loop invariants are manually annotated to the program; we either input these to Frama-C or automatically transform the annotations into witnesses and try to validate these witnesses using CPAchecker's k-induction validator (with k fixed to 1); the listed numbers correspond to the number of successful proofs in each of the sub-folders; we also list the number of successful proofs if no invariants are provided to the tools

Subfolder	Tasks	Frama-C		k-induction	
		with invs.	without invs.	with invs.	without invs.
loop-acceleration	17	3	1	11	4
loop-crafted	2	0	0	2	2
loop-industry-pattern	1	0	0	1	1
loop-invariants	8	3	0	8	0
loop-invgen	5	0	0	2	0
loop-lit	11	6	0	10	2
loop-new	5	1	0	5	2
loop-simple	6	6	0	1	1
loop-zilu	20	9	0	19	7
loops	23	13	6	17	15
loops-crafted-1	27	0	0	12	1
total	125	41	7	88	35

Hoare-style proof. Our benchmark set with manually added ACSL annotations is available in the artifact for this paper [23].[3]

Construction of an Interactive Verifier. With our ACSL benchmark set, we can now convert a witness validator into an interactive verifier as depicted in Fig. 4d. For the validator we use CPAchecker, which can validate witnesses by using the invariants for a proof by k-induction. By fixing the unrolling bound of the k-induction to $k = 1$, this will essentially attempt to prove the program correct via 1-induction over the provided loop invariants. If we do not fix the unrolling bound, the k-induction validation would also essentially perform bounded model checking, so we would not know whether a proof succeeded because of the provided loop invariants or simply because the verification task is bounded to a low number of loop iterations.

Since this 1-induction proof is very similar to what Frama-C's weakest-precondition analysis does, we can directly compare both approaches. As some tasks from the benchmark set do not require additional invariants (i.e., the

[3] Our benchmark set is continuously updated and can also be found at: https://gitlab.com/sosy-lab/research/data/acsl-benchmarks

property to be checked is already inductive) we also analyze how both tools perform on the benchmark set if we do not provide any loop invariants.

The experimental setup is the same described in Sect. 4.1, except that we use a newer version of FRAMA-C-SV in order to use the weakest-precondition analysis of FRAMA-C. The results are shown in Table 3, which lists the number of successful proofs by subfolder. We can observe that both FRAMA-C and our constructed interactive verifier based on CPACHECKER can make use of the information from the annotations and prove significantly more tasks compared to without the annotated loop invariants. This shows that the component described in Fig. 4d is indeed working and useful.

5 Conclusion

The verification community integrates new achievements into two kinds of tools: interactive verifiers and automatic verifiers. Unfortunately, the possibility of cooperation between the two kinds of tools was left largely unused, although there seems to be a large potential. Our work addresses this open problem, identifying witnesses as interface objects and constructing some new building blocks (transformations) that can be used to connect interactive and automatic verifiers. The new building blocks, together with a cooperation framework from previous work, make it possible to construct new verifiers, in particular, automatic verifiers that can be used interactively, and interactive verifiers that can be fed with information from automatic verifiers: Our new program transformations translate the original program into a new program that contains invariants in a way that is understandable by the targeted backend verifier (interactive *or* automatic). Our combinations do not require changes to the existing verifiers: they are used as 'off-the-shelf' components, provided in binary form.

We performed an experimental study on witnesses that were produced in the most recent competition on software verification and on programs with manually annotated loop invariants. The results show that our approach works in practice: We can construct various kinds of verification tools based on our new building blocks. Instrumenting information from annotations and correctness witnesses into the original program can improve the effectivity of verifiers, that is, with the provided information they can verify programs that they could not verify without the information. Our results have many practical implications: (a) automatic verification tools can now be used in an interactive way, that is, users or other verifiers can conveniently give invariants as input in order to prove programs correct, (b) new validators based on interactive verifiers can be constructed in order to complement the set of currently available validators, and (c) both kinds of verifiers can be connected in a cooperative framework, in order to obtain more powerful verification tools. This work opens up a whole array of new opportunities that need to be explored, and there are many directions of future work. We hope that other researchers and practitioners find our approach helpful to combine existing verification tools without changing their source code.

Data-Availability Statement. The witnesses that we used are available at Zenodo [10]. The programs are available at Zenodo [9] and on GitLab at https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp22. We implemented our transformations in the verification framework CPAchecker, which is freely available via the project web site at https://cpachecker.sosy-lab.org. A reproduction package for our experimental results is available at Zenodo [23].

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

Acknowledgment. We thank Nikolai Kosmatov for an inspiring and motivating discussion at the conference ISoLA 2018 on the necessity to combine automatic and interactive verification.

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. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The key tool. Software and Systems Modeling **4**(1), 32–54 (2005). https://doi.org/10.1007/s10270-004-0058-x
3. 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
4. Ayaziová, P., Chalupa, M., Strejček, J.: Symbiotic-Witch: A Klee-based violation witness checker (competition contribution). In: Proc. TACAS (2). pp. 468–473. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_33
5. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with Slam. Commun. ACM **54**(7), 68–76 (2011). https://doi.org/10.1145/1965724.1965743
6. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language version 1.17 (2021), available at https://frama-c.com/download/acsl-1.17.pdf
7. Beckert, B., Hähnle, R.: Reasoning and verification: State of the art and current trends. IEEE Intelligent Systems **29**(1), 20–29 (2014). https://doi.org/10.1109/MIS.2014.3
8. 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
9. 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
10. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5838498
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351

12. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. (2022). https://doi.org/10.1145/3477579
13. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
14. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
15. 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
16. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664
17. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
18. 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
19. Beyer, D., Lemberger, T.: Conditional testing: Off-the-shelf combination of test-case generators. In: Proc. ATVA. pp. 189–208. LNCS 11781, Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_11
20. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2017). https://doi.org/10.1007/s10009-017-0469-y
21. 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
22. Beyer, D., Spiessl, M.: The static analyzer Frama-C in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 429–434. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_26
23. Beyer, D., Spiessl, M., Umbricht, S.: Reproduction package for SEFM 2022 article 'Cooperation between automatic and interactive software verifiers'. Zenodo (2022). https://doi.org/10.5281/zenodo.6541544
24. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
25. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
26. 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
27. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Proc. FM. pp. 132–146. LNCS 7436, Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_13

28. 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
29. Cook, B.: Formal reasoning about the security of Amazon web services. In: Proc. CAV (2). pp. 38–47. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
30. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
31. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE. pp. 100–114. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_7
32. Ernst, G.: A complete approach to loop verification with invariants and summaries. Tech. Rep. arXiv:2010.05812v2, arXiv (January 2020). https://doi.org/10.48550/arXiv.2010.05812
33. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: Overview and VerifyThis competition. Int. J. Softw. Tools Technol. Transf. **17**(6), 677–694 (2015). https://doi.org/10.1007/s10009-014-0308-3
34. Ernst, G., Huisman, M., Mostowski, W., Ulbrich, M.: VerifyThis: Verification competition with a human factor. In: Proc. TACAS. pp. 176–195. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_12
35. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: Proc. FMICS. pp. 3–69. LNCS 12327, Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1
36. Gerrard, M.J., Dwyer, M.B.: Comprehensive failure characterization. In: Proc. ASE. pp. 365–376. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115649
37. Gerrard, M.J., Dwyer, M.B.: ALPACA: A large portfolio-based alternating conditional analysis. In: Atlee, J.M., Bultan, T., Whittle, J. (eds.) Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019. pp. 35–38. IEEE / ACM (2019). https://doi.org/10.1109/ICSE-Companion.2019.00032
38. Hales, T.C., Harrison, J., McLaughlin, S., Nipkow, T., Obua, S., Zumkeller, R.: A revision of the proof of the Kepler conjecture. Discret. Comput. Geom. **44**(1), 1–34 (2010). https://doi.org/10.1007/s00454-009-9148-4
39. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
40. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **1610**(06229) (October 2016)
41. Heule, M.J.H.: Schur number five. In: Proc. AAAI. pp. 6598–6606. AAAI Press (2018)
42. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D., Păsăreanu, C.S.: Rigorous examination of reactive systems. The RERS challenges 2012 and 2013. Int. J. Softw. Tools Technol. Transfer **16**(5), 457–464 (2014). https://doi.org/10.1007/s10009-014-0337-y
43. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. NFM. pp. 41–55. LNCS 6617, Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
44. Jakobs, M.C., Wehrheim, H.: Certification for configurable program analysis. In: Proc. SPIN. pp. 30–39. ACM (2014). https://doi.org/10.1145/2632362.2632372

45. Jakobs, M.C., Wehrheim, H.: Programs from proofs: A framework for the safe execution of untrusted software. ACM Trans. Program. Lang. Syst. **39**(2), 7:1–7:56 (2017). https://doi.org/10.1145/3014427
46. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proc. LPAR. pp. 348–370. LNCS 6355, Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
47. Necula, G.C.: Proof-carrying code. In: Proc. POPL. pp. 106–119. ACM (1997). https://doi.org/10.1145/263699.263712
48. Richter, C., Hüllermeier, E., Jakobs, M.-C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
49. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: The Goblint approach. In: Proc. ASE. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337
50. Švejda, J., Berger, P., Katoen, J.P.: Interpretation-based violation witness validation for C: NitWit. In: Proc. TACAS. pp. 40–57. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_3