The Transformation Game: Joining Forces for Verification

Dirk Beyer and Nian-Ze Lee

LMU Munich, Munich, Germany

Abstract. Transformation plays a key role in verification technology, conveying information across different abstraction layers and underpinning the correctness, efficiency, and usability of formal-methods tools. Nevertheless, transformation procedures are often tightly coupled with individual verifiers, and thus, hard to reuse across different tools. The lack of modularity incurs repeated engineering effort and the risk of bugs in the process of 'reinventing the wheel'. It can be seen as a new paradigm to construct verification technology by employing standardized formats and interfaces for information exchange, and by building modular transformers between verification artifacts. Following this paradigm of *modular transformation*, recent works have (1) enhanced and complemented the state of the art by transforming verification tasks and applying tools for other modeling languages or specifications, (2) built new tools by combining mature ones via standardized formats for exchanging verification artifacts, and (3) communicated certificates of verification results to improve usability and explainability. In this paper, we survey existing transformation procedures and advocate the paradigm of modular transformation and exchange formats. Our vision is an ecosystem of reusable verification components that supports joining forces of all available techniques, allows agile development of new tools, and provides a common ground to evaluate and compare future scientific advancements: via modular transformation.

Keywords: Formal verification · Model checking · Transformation · Standard exchange format · Intermediate representation · Verification witness · Component-based design · Modularity · Quality

1 Introduction

Formal verification, including model checking [1], is an important technology to examine with mathematical rigor whether a computational model satisfies a given specification. It has attracted significant attention not only in academia but also in industry and helped to find potential errors or delivered correctness guarantees of practical computing systems [2, 3, 4, 5, 6, 7]. The challenges for formal verification in the modern era have escalated due to multiple reasons, including the ever-increasing system complexity, emerging applications with domain-specific modeling languages, interactions among heterogeneous computing components, and the usage of generative artificial intelligence to aid the implementation of

systems. Therefore, formal-methods researchers and practitioners have to consider the following essential question: **How can we continue to improve existing tools and develop new tools to tackle the aforementioned challenges while keeping up with the ever-increasing development pace?**

To approach this question, we observe that formal-methods tools heavily rely on *transformation*. For example, consider the following three-step approach: a model in a frontend modeling language and a specification are transformed to an intermediate representation used by a tool, the semantics of the intermediate representation is transformed to mathematical formulas in the logic used by a prover, and the answer of a prover is transformed back to a certificate in the frontend language to help users to understand the verification results. Various transformation procedures connect different abstraction layers in formal-methods tools and play a key role in their correctness, efficiency, and usability. This 'transformation game' can be made explicit, and the steps could be decomposed such that the components implementing each step can be reused.

Technologies that can be used as library, such as SAT [8] and SMT [9] solvers, usually have standardized input formats and application programming interfaces to facilitate their usage by formal-methods tools in an off-the-shelf manner. On the contrary, the transformation procedures in verification tools are often integrated and hard-wired in individual tools. The lack of modularity and reusability makes the development of formal-methods tools error-prone, because developers often need to re-implement the same techniques. For example, a new verifier for the programming language C [10] often needs to implement the functionality to interpret the semantics of C [11] (e.g., as logical formulas in SMT-LIB 2 format [12]) and can hardly reuse the transformation procedures in other mature software verifiers. This slows down the progress pace in the development of verification tools and the growth of the importance of formal methods. The goal of this paper is to advocate transformation as a means to decompose verification tools.

1.1 Transformation as a Paradigm to Develop Formal-Methods Tools

We advocate the possibility to construct verification tools by means of transformation, to continue the success story of formal verification while facing the contemporary challenges. The paradigm of *modular transformation* calls for using or inventing exchange formats to represent verification artifacts, building modular transformers based on the exchange formats, and employing transformers as building blocks to develop new tools.

A related paradigm to modular transformation is *cooperative verification* [13], which advocates that several tools should work together to solve a task by exchanging information. A framework has been developed to facilitate the combination and collaboration of tools [14]. Modular transformation differs from cooperation verification in its emphasis on the transformation components of verification tools. While cooperative verification investigates how different tools can solve a verification task together, modular transformation deals with the design of transformation components such that they are maximally reusable and can be easily combined to construct new verification tools, with or without cooperation. A major goal of

modular transformation is to decompose tightly integrated monolithic tools for modularity. The following two ingredients are essential to achieve the goal:

1. *standard exchange formats* for communicating via verification artifacts and
2. *composable standalone transformers* based on verification artifacts.

The terms *verification artifact* and *transformer* are taken from previous work on cooperative verification [13], and we will define and extend them for modular transformation in Sect. 2.

Modular transformation follows the well-known principles of *single responsibility* [15] and *separation of concerns* in component-based software engineering [16]. The merits of these principles have been witnessed in different application domains, including various utility programs in the Unix operating system [17] (i.e., the *Unix philosophy* [18]) and numerous dialects and translators in the LLVM/MLIR compilation infrastructure [19, 20]. In the area of formal verification, research works that follow the paradigm of modular transformation have contributed to

1. combine multiple tools via standard exchange formats for verification [21, 22], validation [23, 24], and testing [25, 26],
2. advance the state of the art by transforming verification tasks and applying tools or algorithms for other modeling languages [27, 28, 29] or specifications [30, 31, 32], and
3. enhance the explainability of verification results by transforming certificates [33, 34, 35].

1.2 Our Vision: Composition through Transformation

How can modular transformation help to address the modern challenges of formal verification? Using the verification of emerging computational models as an example, we illustrate the paradigm's potential below. Suppose we need to build a new verifier for computational models described in a domain-specific language (DSL), which is unsupported by any existing formal-methods tool. To leverage mature verification technology, for example, verifiers for the programming language C, we develop a standalone transformer to translate the DSL to the programming language C. Following the modular transformation paradigm, a new verifier for the DSL can be constructed by combining the translator with state-of-the-art verifiers for C programs in an off-the-shelf manner. This strategy uses the C language as the exchange format to connect the translator with verifiers for C programs and saves the effort to repeatedly implement a frontend for the DSL in every C verifier. Of course, it is also possible to develop a dedicated verifier from scratch that works directly on the DSL. This approach may leverage specific characteristics of the DSL and yield more powerful verifiers. However, the composition of the translator and existing, highly tuned verifiers for C programs can serve as baseline for performance comparison. For example, a recent study [27] compares dedicated verifiers for the BTOR2 language [36], a DSL to describe model-checking problems of hardware circuits, to compositional verifiers formed by a BTOR2-to-C translator and verifiers for C programs. While the dedicated BTOR2 verifiers are better at

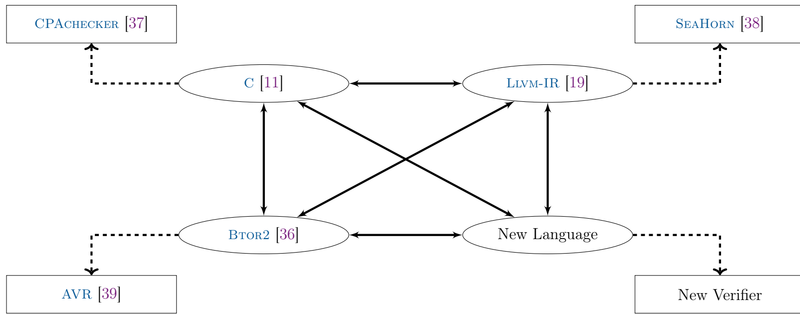

Fig. 1: Transformation network for joining forces via translating verification tasks

proving the safety of hardware circuits, the compositional verifiers reveal bugs in hardware circuits that the dedicated verifiers overlook.

Next we explain how *transformation networks* bring us closer to the modular transformation paradigm, and how to *join forces for formal verification*. We illustrate this in Fig. 1 on the programming language C, the Llvm intermediate representation [19], and the hardware modeling language Btor2 [36] as examples. The transformation network bridges the gaps between various verification communities by translating frontend modeling languages and applying all available backend tools to verify a task. For example, a hardware circuit in the Btor2 language can be translated to a program in C or the Llvm [19] intermediate representation (IR) and given to verifiers for C, e.g., CPAchecker [37], or Llvm, e.g., SeaHorn [38], respectively. A program in C or Llvm can also be translated to a Btor2 circuit and verified by hardware model checkers, such as AVR [39].

In the transformation paradigm, an emerging new modeling language can leverage state-of-the-art verifiers for different languages once the corresponding translators are constructed, which also provides a common ground to evaluate future research advancements. Thanks to the emphasis on standard exchange formats and interfaces in the transformation paradigm, developing new tools as combinations of existing ones becomes easier. The principles of single responsibility and separation of concerns facilitate the participation of community members in the implementation and maintenance of transformers because they are compact and extensible.

While transformation procedures frequently appear in the verification literature, the modular transformation paradigm encourages to decompose the verification into modular components that are easier to develop and maintain. The scope of transformation procedures extends beyond models and specifications to certificates, invariants, and counterexamples. The modular transformation paradigm can significantly increase our performance in developing new verification tools: It will guide us to achieve a more reusable and robust infrastructure for formal verification, avoid bugs in the process of reimplementing standard verification components, and facilitate combination and cooperation between tools through standard exchange formats and interfaces (cf. community discussion [40, 41]).

We humbly contribute this article to the Festschrift on the occasion of Joost-Pieter Katoen's 60th birthday, in order to trigger discussion in the community and draw attention to the modular-transformation paradigm as a potential solution to the modern challenges of formal verification. The rest of this article is organized as follows: Section 2 defines verification artifacts and transformers, Sect. 3 surveys various transformers, either modular or tightly integrated ones, in formal-methods tools, Sect. 4 highlights the construction of new verification tools using transformers, Sect. 5 discusses prospective benefits and impacts of the modular-transformation paradigm, before we conclude in Sect. 6.

2 Verification Artifacts and Transformers

As foundation for the discussion in the rest of this paper, we define *verification artifacts* and *transformers* in this section. The definitions below refine the notions used in the context of *cooperative verification* [13].

Definition 1. *A verification artifact is (1) an input, output, or intermediate object consumed or produced by a formal-methods tool and (2) can be represented in an exchange format defined for communicating information among the tool's internal components or with other formal-methods tools.*

Note that a verification artifact is required to be representable in an exchange format devised for *verification purposes*. Although exchange formats do not need to be human-readable (e.g., the binary AIGER format [42] for hardware model checking), serializing an object to a byte stream does not qualify the object as a verification artifact because serialization protocols such as Java Object Serialization Specification are not designed specifically for formal verification.

Table 1 lists the most common and important types of verification artifacts. A *model* $M \in \mathcal{M}$ is a description of the system under verification. It can be written in a high-level modeling language used by human designers or an IR used by tools. A *specification* $\varphi \in \Phi$ defines the expected behavior of the system under verification. A model M and a specification φ jointly form a *verification task*, which is given to a formal verifier as input. A *verdict* $r \in \mathcal{R}$ is a verifier's decision on a verification task. A verifier may produce three different verdicts: `TRUE`, meaning that $M \models \varphi$; `FALSE`, meaning that $M \not\models \varphi$; and `UNKNOWN`, meaning that the verification was inconclusive. A *witness* $\omega \in \Omega$ is a certificate produced by a tool to explain its verdict on a verification task. We interpret witnesses more flexibly than *verification witnesses* [43] for verifiers and also consider *test cases* as witnesses from test-case generators. Tools that produce witnesses for their analysis results are said to be *certifying* [44, 45]. A *verification condition* $vc \in \mathcal{VC}$ is an intermediate object produced by a verifier to encode a partial or the complete behavior of the model as a set of constraints in the logic of an underlying prover used to decide the satisfiability of vc.

Definition 2. *A transformer is a procedure that consumes one or more verification artifacts as input and produces one or more verification artifacts as output in polynomial time.*

Table 1: Examples of important verification artifacts

Type	Notation	Usage
Model	$\mathcal{M}$	Description of the system under verification
Specification	Φ	Expected behavior of the system under verification
Verdict	$\mathcal{R}$	Decision on whether a model satisfies a specification
Witness	Ω	Certificate explaining the verdict of a tool
Verification condition	$\mathcal{VC}$	Set of constraints that encode the behavior of a model

To capture a transformer's essence of being simple and fast compared to a prover or a verifier (called *analyzer* in cooperative verification [13, 14]), we require it to produce output artifacts in time polynomial in the size of the input artifacts. The definition can be relaxed in other scenarios, e.g., when a transformation component runs faster than the main verification process but not in polynomial time.

Table 2 lists the most common and important types of transformers. A *translator* consumes a model M and produces a behaviorally equivalent model M' in a *different* modeling language. An *encoder* consumes a model M and produces a verification condition vc that describes partial or complete behavior of M. A *specification transformer* consumes a model M and a specification φ and produces a transformed model M' and a different specification φ' such that $M \models \varphi \iff M' \models \varphi'$ (i.e., the two verification tasks are equisatisfiable). A *witness transformer* consumes a model M and a witness ω and produces a transformed witness ω' for other purposes, e.g., by making it more precise to replay a counterexample. A *slicer* cuts off some parts of the consumed model M that do not affect the satisfiability of the specification φ by M and produces a sliced model M'. A *splitter* decomposes a verification task (M, φ) into smaller tasks, where the number of split tasks is polynomial in the sizes of M and φ. A *pruner* removes irrelevant parts of the consumed model M according to a witness ω and produces a simplified model M' (e.g., by deleting paths marked as fully explored in ω from M). An *annotator* augments the consumed model M according to a witness ω and produces an augmented model M' (e.g., by labeling the invariants recorded in ω as assumptions to M). A *reducer* consumes a model M and produces a model M' that is in the *same* modeling language and easier to verify. An *instrumentor* expands a model M and produces a model M' in the *same* modeling language to record information for further analysis, e.g., by adding run-time monitors to M.

The key difference between a pruner and a reducer (resp. an annotator and an instrumentor) in the above classification is the presence of a witness as an input to the transformer. A pruner or an annotator relies on the information in the witness to transform the input model, whereas a reducer or an instrumentor transforms the model without taking a witness into account.

Table 2: Examples of important transformers

Type	Signature	Functionality
Translator	$\mathcal{M} \mapsto \mathcal{M}$	Translates a model to a behaviorally equivalent one in a different language
Encoder	$\mathcal{M} \mapsto \mathcal{VC}$	Describes partial or complete behavior of a model as a verification condition
Specification transformer	$\mathcal{M} \times \Phi \mapsto \mathcal{M} \times \Phi$	Converts a verification task to an equisatisfiable one with a different specification
Witness transformer	$\mathcal{M} \times \Omega \mapsto \Omega$	Transforms a witness for a model to another witness, e.g., by making it more precise
Slicer	$\mathcal{M} \times \Phi \mapsto \mathcal{M}$	Removes some parts of the model that do not affect the satisfiability of the specification
Splitter	$\mathcal{M} \times \Phi \mapsto (\mathcal{M} \times \Phi)^+$	Decomposes a verification task into a polynomial number of smaller tasks
Pruner	$\mathcal{M} \times \Omega \mapsto \mathcal{M}$	Removes irrelevant or fully-explored parts of a model based on a witness
Annotator	$\mathcal{M} \times \Omega \mapsto \mathcal{M}$	Augments (e.g., annotating invariants) a model based on a witness
Reducer	$\mathcal{M} \mapsto \mathcal{M}$	Simplifies a model (in the same language) so that it is easier to verify
Instrumentor	$\mathcal{M} \mapsto \mathcal{M}$	Expands a model (in the same language) to record information for further analysis

3 Transformers in Formal-Methods Tools

In this section, we survey transformation approaches and tools developed for formally verifying hardware and software computational models. Following the classification of transformers discussed in Sect. 2, our survey aims to cover representative transformers of different types in the literature. Table 3 lists the transformers discussed in this section (and the tools constructed by using transformers, which will be discussed in Sect. 4).

3.1 Translators

A translator consumes a model as input and produces a behaviorally equivalent model in a different modeling language. We first look at translators whose outputs are in a high-level language used by human engineers to design the system and then discuss translators whose outputs are an intermediate representation used in or consumed by other tools.

Translators can also be used to construct analyzers for the source language via their combinations with tools for the target language as backend [27, 28, 33, 46]. Such translators can help with comparing and bridging verification approaches for different computational models and will be covered in Sect. 4.1.

To High-Level Modeling Languages. Models for digital sequential circuits can be represented in hardware-description languages at the register-transfer level, such as Verilog [47]. A Verilog circuit can be translated into a behaviorally equivalent C program to leverage analysis techniques for software. For example, Verilator [48] translates Verilog circuits to multi-threaded C++ programs, which can be compiled and executed for efficient circuit simulation. VHD2CA [49] translates VHDL [50] circuits to counter automata, which can be dumped as C programs and checked by software analyzers.

To Intermediate Representations. Intermediate representations are used by tools to exchange information among their internal components or with external tools. They usually have a simple syntax to facilitate parser development of the tools and precise semantics to underpin formal reasoning of the models. Many IRs have been proposed as the input format for formal verifiers. For example, SMV [51] is a language to describe finite-state transition systems and allows properties to be specified in CTL; AIGER [42] is used to describe a bit-level *and-inverter graph* with memory elements; Btor2 [36] lifts AIGER to the word level by inheriting operations from the SMT-LIB 2 format [12]. These three formats are mainstream in the community of hardware model checking and often used as target languages by translators. For example, Verilog2SMV [52] and Ver2Smv [53] translate a Verilog circuit to an SMV model, which can be given to nuxmv [54] for verification. A word-level circuit in the Btor2 language can be bit-blasted to an AIGER circuit by Btor2AIGER from the Btor2 tool suite [55] to leverage bit-level model checkers, such as ABC [56]. CIRCT [57] is an open-source tool chain for electronic design automation based on the MLIR [20] framework, offering various IRs and tools to represent and translate hardware circuits.

In the community of software analysis, Boolean programs [58] and Goto programs have been used by pioneering verifiers, e.g., Slam [2, 59] and Cbmc [60], as the internal problem representation. To facilitate the analysis of C code, Cil [61], the C Intermediate Language, decomposes complex constructs of C into simpler ones but still works at a higher level than the assembly code. Since the invention of Llvm [19], some software analyzers rely on its rapidly-growing compilers, including Clang, as frontend to translate the source programming language to Llvm-IR, used as the internal data structure. For example, Llbmc [62] and SeaBmc [63] translate C to Llvm-IR and perform bit-precise BMC with SMT solving. Klee [64] is an Llvm-based symbolic-execution engine for test-case generation. Symbiotic [65, 66] extends Klee with program slicing and instrumentation to implement a full-fledged software verifier. SeaHorn [38] uses Clang as its frontend and generates verification conditions from Llvm-IR. Smack [67] translates Llvm-IR to Boogie [68], an intermediate verification language, and relies on verifiers for Boogie as backend. MLIR [20] is an extension of Llvm to flexibly define "dialects" of Llvm-IR and automatically generate translators between the dialects. In a similar spirit, the software verifier Infer [5] uses an IR called SIL [69] to decouple the frontend and backend development.

3.2 Encoders

An encoder consumes a model M and produces a verification condition vc that describes partial or complete behavior of M in terms of the logic used by some prover. Commonly used logics for formal verification include propositional logic [8], first-order logic with background theories [9], and constrained Horn clauses (CHC) [70], a fragment of first-order logic. Formulas in propositional logic are described in DIMACS format [71] and given to Boolean Satisfiability (SAT) solvers. Formulas in first-order logic modulo some background theory are described in SMT-LIB 2 format [12] and given to Satisfiability Modulo Theory (SMT) solvers that support the theory. CHC formulas are also described in SMT-LIB 2 format and given to Horn solvers [72, 73, 74, 75].

While prover-based verifiers have an encoding procedure from their IRs to the logic of the prover, the procedure is often tightly integrated with the verifiers and hardly reusable in other tools. We highlight several tools that can dump verification conditions and be used as standalone encoders. EBMC [76] takes as input a Verilog circuit or an SMV model, unrolls the model, then generates verification conditions for the behavior of the unrolled model as SAT or SMT formulas. Llbmc [62] and SeaBmc [63] consume a program in Llvm-IR and produce formulas for bounded model checking as SMT formulas. SeaHorn [38] consumes Llvm-IR and produces verification conditions as CHC formulas. Boogie [68] and Viper [77] are intermediate languages for verification, accompanied by encoders from a Boogie or Viper program to SMT formulas.

Besides the logics discussed above, linear programming and its variants have also been used to encode verification conditions for analyzing software programs [78] or neural networks [79, 80].

3.3 Specification Transformers

A specification transformer consumes a verification task (M, φ) and produces another verification task (M', φ') such that (1) $\varphi \neq \varphi'$ and (2) $M \models \varphi \iff M' \models \varphi'$. Since reachability-safety analysis has received much (if not most) attention, and many mature tools are available, several approaches have been invented to transform verification tasks with other specifications to reachability safety.

Schuppan and Biere proposed a transformation for finite-state transition systems with a liveness specification [30]. The key observation is that a counterexample to a liveness specification in a finite-state model has a "lasso" shape, which consists of a prefix of finite length leading to the entry of a loop. By enlarging the state space of the original model with a shadow variable to record visited states and introducing an additional input as an oracle to detect the entry of a lasso's loop, the approach transforms a verification task with a liveness specification to another task with a reachability-safety specification to find a lasso.

CCured is a type system to infer the memory safety of a program [81]. For pointers whose safety cannot be statically inferred, CCured inserts run-time checks in the original program, which can be used as error locations and checked by model checkers, such as Blast [31].

Lal and Reps proposed a transformation from programs with a concurrency-safety specification to sequential programs with a single-thread reachability-safety specification [32]. The main idea is to limit the number of context switches in the concurrent program and simulate the behavior of the concurrent program with a sequential program.

Testability transformation [82] studies mappings from a model M and a *coverage specification* (also called test-adequacy criterion) φ to a model M' and a coverage specification φ' such that for any set T of tests, T is adequate for M w.r.t. φ if and only if T is adequate for M' w.r.t. φ'. Testability transformation is used to facilitate generation of test cases.

It was also suggested to partition a conjunctive specification into its conjoined parts and verify the program several times with smaller specifications [83].

3.4 Witness Transformers

Verification witnesses [43] were introduced as justifications for the verification verdict. In particular, there is no prescribed level of abstraction for verification witnesses. For example, violation witnesses can be very abstract by only avoiding some parts of the program, or they can be as concrete as a test case by assigning explicit values to inputs. The model checker Blast contains a procedure to internally transform a counterexample, which can be seen as a violation witness in the form of a feasible error path, to test cases [84]. We will discuss several witness transformers and how they can be used to combine tools in Sect. 4.2.

3.5 Slicers

A slicer consumes a verification task (M, φ) and produces a model M' by cutting off the parts in M that do not affect its satisfaction of φ. For hardware model checking, slicing computes the *cone of influence* of the asserted signals and removes the circuitry outside the cones from the original circuit. For software verification, slicing [85] removes program operations that do not touch variables on which the assertions depend directly or indirectly.

Slicing has been shown to be an effective technique for software verification [86]. The tool Symbiotic [65, 66] applies slicing and instrumentation as modular transformations, before a symbolic-execution backend like Klee [64] is asked to perform a reachability analysis.

A project by Joost-Pieter Katoen studied the application of tools for software model checking to automotive code [87]. This project also investigated the impact of slicing, and confirmed the conclusion from the above study on Symbiotic. Slicing had a significant impact on the performance and effectiveness of several software verifiers [88, Sect. 4.8 “The Effect of Static Analysis”].

3.6 Splitters

A splitter decomposes a verification task (M, φ) into several smaller tasks such that (1) $M \models \varphi$ can be derived from the verification results of the smaller

tasks and (2) the number of the smaller verification tasks is polynomial in the sizes of M and φ.

Splitting was first studied for logic programs to decompose them into a bottom part and a top part to simplify the computation of *answer sets* [89]. It has been used also to scale symbolic execution by splitting program paths into several *ranges* (i.e., sets of paths) and exploring path ranges in parallel [90]. *Ranged symbolic execution* can be further generalized to run different analyses on different ranges [91]. Splitting has also been used to accelerate data-flow analysis by decomposing a control-flow graph into several subgraphs, applying data-flow analysis to each subgraph, and combining the resulting invariants from each subgraph [92, 93].

A program can also be decomposed into a set of straight-line programs that correspond to the verification conditions of a Hoare-style proof of correctness. This strategy has been implemented in the verification tool chain VST-T [94]. Later the same strategy has been used by the witness validator LIV [24], which decomposes a C program into a set of C programs that can be verified with off-the-shelf verifiers for C. Also the verifier Bubaak-Split [95] splits programs into parts that can be verified in parallel. However, it is not clear if Bubaak-Split's splitting can be done in polynomial time, because it involves a lightweight verification pass.

3.7 Pruners and Annotators

Pruners and annotators are dual to each other. While they share the same signature $\mathcal{M} \times \Omega \mapsto \mathcal{M}$, the former use the information in witnesses to remove or simplify irrelevant parts of the model, and the latter add hints to the model for further analyses.

For example, data-flow analysis can be combined with a pruner that transforms the control-flow graph [96]. Propagating variables with constant values discovered in data-flow analysis (which can be stored as a witness), the pruner simplifies the original control-flow graph to a subgraph to speedup the analysis. In Sect. 4.3, we will discuss how a pruner [22] can be used to turn any off-the-shelf model checker into a conditional model checker [97].

Program annotators have been used to coordinate static model checkers and test-case generators by explicitly documenting the assumptions made by model checkers in the program under verification to guide test-case generators, such as symbolic execution, to explore unverified parts of the program [98, 99]. They have also been used to orchestrate automatic and interactive verifiers [35, 100]. More details will be discussed in Sect. 4.3.

3.8 Reducers and Instrumentors

Reducers and instrumentors are dual to each other. While they share the same signature $\mathcal{M} \mapsto \mathcal{M}$, the former simplify the model to make it easier to verify, and the latter expand the model by adding functionalities, e.g., run-time monitors, to record information for further analyses. In contrast to reducers, instrumentors usually slightly increase the workload for verifiers.

Acceleration [101, 102], *shrinking* [103], and *abstraction* [104, 105, 106] are three important techniques to reduce programs with loops. Loop acceleration replaces a loop with its transitive closure to speed up the convergence to a fixed point. Loop shrinking reduces a loop that processes a large array to an overapproximating but much smaller loop, which can be handled by bounded model checking. Loop abstraction overapproximates a loop by, for example, havocing variables modified in the loop. The key idea behind the three reducers is to produce a program with simplified computation in loops that overapproximates the original program, such that the correctness of the simplified program (which is easier to prove) implies the correctness of the original program.

Program instrumentors add auxiliary functionalities to a model to record more information for further analyses. For example, Symbiotic [65, 66] checks memory safety by instrumenting C code that tracks allocated memory regions to the original program [107].

4 Constructing Formal-Methods Tools using Transformers

In this section, we discuss how the modular-transformation paradigm outlined in Sect. 1.1, which emphasizes standardized exchange formats for verification artifacts and modular transformers, can help to construct new formal-methods tools in an agile and reliable way by combining existing ones with transformers. We will highlight tool combinations using translators and witness transformers, because they can help to unify and join forces of various verification techniques, such as hardware model checking vs. software verification and automatic model checking vs. interactive verification.

To facilitate tool combination, CoVeriTeam [14] is a domain-specific language and tool to combine off-the-shelf executable components. Transformers can be composed with other components to implement a composite verification tool. For example, it was used to compose several components to construct tools using algorithm selection and portfolios [108]. CoVeriTeam has also been used to implement approaches like ranged analysis [91], circuit-based program verification [29], cooperative test-case generation [109], conditional testing [25], and component-based CEGAR [21].

4.1 Combining Tools with Translators

As mentioned in Sect. 1, one challenge to formal verification in the modern era is to handle systems with heterogeneous computational models, including hardware circuits, software programs, and cyber-physical devices. Although formal methods for different computational models share similar concepts and techniques, their alignment with distinct modeling languages creates gaps between the research communities. These gaps hinder one community from enjoying the advancements of the others, and the potential of leveraging techniques developed in related areas to improve the state of the art may be overlooked. Transformers, especially translators, can help to bridge the gaps between communities and provide a common ground to compare approaches from different communities.

For example, the tool v2c [110] translates a Verilog circuit to a cycle-accurate and bit-precise C program, which can be consumed by software verifiers to analyze the properties of the Verilog circuit. A circuit model can also be represented in an intermediate representation designed for verification, e.g., the Btor2 modeling language [36], the input format in the Hardware Model-Checking Competitions (HWMCC) [111, 112]. The tool Btor2C [27] translates a Btor2 model to an equivalent C program, enabling head-to-head comparison between hardware model checkers from HWMCC and software verifiers for C programs in the Competitions on Software Verification (SV-COMP) [113]. The evaluation shows that software verifiers can detect more hardware bugs than mature hardware model checkers [27]. Similarly, Btor2MLIR [28] defines a dialect for Btor2 in the MLIR [20] framework and translates a Btor2 model into Llvm-IR. Software analyzers consuming Llvm-IR, such as SeaHorn [38], Smack [67], and Klee [64], can then be used to examine properties of Btor2 circuits.

Translation from the other direction, i.e., from software to hardware, has also been investigated. For example, c2v [114] translates a C program to a Verilog circuit by first compiling it to Llvm-IR [19] and translating the IR to Verilog. Hardware model checkers can be applied to the translated circuit to analyze properties of the original C program. The model checker Kratos2 [115] uses the K2 language as its intermediate representation for C programs and can be used as a translator to fold a C program into a sequential circuit in SMV, Btor2, or AIGER formats. Using Kratos2 as a translator, the software verifier CPV [29] employs award-winning verifiers in HWMCC, such as ABC [56] and AVR [39], as its backend and participated in SV-COMP 2024. As a first-time participant, CPV performed better than many mature and fine-tuned software verifiers, indicating the potential of using hardware model checkers for program analysis.

VMT [116] and MoXI [46, 117] are two IRs for infinite-state transition systems. They both add constructs to define a transition system on top of the SMT-LIB 2 format [12] and offer translators from various modeling languages to themselves and vice versa. Their goal is to provide a common ground for comparing formal-methods tools and exchanging verification tasks: A verification task can be translated to other modeling languages through them and solved by tools from other domains. Tools for different modeling languages can be compared directly by using their translators as preprocessing. Recently, the first direct model checker for MoXI, MoXIchecker [118], has been proposed, which supports various background theories to describe model-checking problems without translating them to other modeling languages.

4.2 Combining Tools with Witness Transformers

Witnesses can be transformed to improve the explainability and usability of analysis results. For example, a more abstract violation witness can be *testified* step by step to a more concrete one [119]. CPA-witness2test implements the approach of *execution-based validation* [34, 84] and transforms violation witnesses to test cases. With the help of CPA-witness2test, witness-generating software verifiers can be combined with debuggers to exempt programmers from figuring out how to

interpret violation witnesses and maximize the benefits of formal-methods tools in a standard software-development environment. Note that, while *stepwise testification* may not run in polynomial time,[1] it is usually much faster than the verification process, and hence we also consider stepwise testification as a transformer.

Witness transformers can also convert the information of witnesses produced by verifiers for different modeling languages to other domains. For example, a standalone witness transformer from automata-based software witnesses [43] to hardware witnesses in the Btor2 language [36] is used to combine software verifiers and Btor2 witness validators into a certifying hardware model checker Btor2-Cert [33]. The software verifier CPV [29] also has a witness transformer from Btor2 witnesses to software witnesses to combine hardware model checkers and software witness validators.

Witness transformers can also be used to combine automatic and interactive software verifiers [35]. Invariants found by automatic verifiers and recorded in correctness witnesses can be transformed to program annotations, e.g., in the ACSL language [120], and verified by interactive verifiers like Frama-C [121]. Vice versa, the proof obligations discharged by an interactive verifier can be transformed to a correctness witness, and an automatic witness validator can be invoked to check the proof obligations.

4.3 Combining Tools with Other Transformers

Transformers can be used to combine tools to cooperatively solve a complex problem via leveraging their unique strengths. For example, *conditional model checking* for software verification [97] uses *condition automata* (which can be seen as witnesses in our categorization) to mark the already-explored paths in a program and instruct subsequent verifiers to work on unexplored paths. To be used for conditional model checking, a verifier needs to take a condition automaton into account during its analysis, which incurs extra engineering effort and hinders a wider adoption of the idea.

Following the modular transformation paradigm, researchers proposed a standalone **pruner** to remove the explored parts from a program based on a condition automaton, which produces a *residual program* that can be consumed directly by off-the-shelf verifiers [22]. Using this pruner for preprocessing, any verifier can become a new conditional model checker without extra engineering effort. A similar **pruner** has also been developed for test-case generators [25], where already-hit test goals are removed from the original program by the pruner to form a residual program for subsequent test-case generation runs.

Transformers also enable verifiers to be used as building blocks for other applications beyond their original use cases, such as test-case generation and witness validation. Using a **specification transformer** to convert test goals to error locations for a reachability-safety specification, CoVeriTest [26] generates test cases by invoking off-the-shelf software verifiers to find counterexamples that

[1] E.g., when the testifier resorts to satisfiability solving to refute a violation witness without feasible error paths.

reach the error locations (i.e., the test goals). The witness validator MetaVal [23] validates verification witnesses by using an **annotator** to label the information in a witness as assumptions (to guide the counterexample search for violation witnesses) or assertions (to check the invariants for correctness witnesses) in the original program and invoking software verifiers to analyze the annotated program. To validate correctness witnesses, LIV [24] uses an **annotator** to label a program with invariants and a **splitter** to decompose the annotated program into several straight-line subprograms. An off-the-shelf verifier is invoked on the split programs to check the inductivity of the invariants.

Transformers can also benefit or facilitate the development of verifiers. For example, to leverage techniques for loop acceleration [101, 102], shrinking [103], or abstraction [104, 105], developers of verifiers often need to implement the successful approaches in their own tools again. To mitigate the risk of bugs for reinventing the wheel, CEGAR-PT [106] is a standalone **reducer** that implements various abstraction techniques at the source-code level and can be used by other verifiers for program transformation. A transformed program is returned as an ordinary input program to a verifier and can be readily analyzed. To facilitate transforming witnesses produced by hardware model checkers to witnesses for programs, the software verifier CPV [29] uses an **instrumentor** to augment input C programs before verification.

5 Applications of Transformation to Construct Modular Tools: Benefits and Impact

Table 3 summarizes the transformers and tools using transformers discussed so far in this paper. The collected entries in the table are not meant to be comprehensive because they are limited by the space in this article and by our knowledge. The table is an initial attempt to draw the community's attention to the problem of decomposing verification tools by making transformation modular and explicit. We also call out the members in the verification community to contribute and maintain a data base for various transformers and verification tools. Such an effort has been started recently with a collection of tools for formal methods [122, 123], and we have added the used transformers to the meta data of the tools that are available there. In the following, we would like to mention a few example approaches of modular transformation together with their benefits and potential impact.

Modular Construction of Verification Tools. Modular construction from components ideally supports agile development processes, which are focussed on increments. If the solution consists of several independent components, the development can also be more independent. This has also two immediate consequences: First, a system built from loosely coupled components (ideally independent executables) is easy to extend, due to the reduced set of preconditions (such as having to use the same programming language for implementation). Second, a system that is modularly built using transformers can be more reliable, because the components might be developed, used, tested, and improved by different and large user groups.

Table 3: Transformers and formal-methods tools using transformers (column "Type" is the type of the transformer used by the tool)

Approach/Tool	Type	Description	Ref.
Btor2AIGER	Translator	From Btor2 to AIGER for verification	[55]
Btor2C	Translator	From Btor2 to C for verification	[27]
Btor2MLIR	Translator	From Btor2 to Llvm-IR for verification	[28]
C2BP (part of Slam)	Translator	From C to a Boolean program for verification	[2, 58, 59]
c2v	Translator	From C to Verilog for verification	[114]
Cil	Translator	From C to Cil for analysis	[61]
CIRCT	Translator	Compiler from various hardware description languages to hardware IRs	[57]
Cbmc (part of CProver)	Translator	From C to a Goto program for verification	[60]
Infer	Translator	From Java, C, C++ to SIL for verification	[5, 69]
Kratos2	Translator	From C to K2 language for verification	[115]
Llvm	Translator	Compiler from various programming languages to Llvm-IR	[19]
MLIR	Translator	Translate Llvm dialects	[20]
MoXI	Translator	Translation between SMV, Btor2, and MoXI	[46, 117]
Smack	Translator	From Llvm-IR to Boogie	[67]
v2c	Translator	From Verilog to C for verification	[110]
Ver2Smv	Translator	From Verilog to SMV for verification	[53]
Verilator	Translator	From Verilog to C++ for simulation	[48]
Verilog2SMV	Translator	From Verilog to SMV for verification	[52]
VHD2CA	Translator	From VHDL to counter automata and C for verification	[49]
VMT	Translator	Translation between SMV, Btor2, and VMT	[116]
Boogie	Encoder	From Boogie programs into SMT formulas	[68]
EBMC	Encoder	From hardware circuits into SAT or SMT formulas	[76]
Llbmc	Encoder	From Llvm-IR into SMT formulas	[62]
SeaBmc	Encoder	From Llvm-IR into SMT formulas	[63]
SeaHorn	Encoder	From Llvm-IR into CHC formulas	[38]

Approach/Tool	Type	Description	Ref.
Viper	Encoder	From Viper programs into SMT formulas	[77]
CoVeriTest	Spec. transformer	From test goals to error locations and use reachability analyzers to generate tests	[26]
CSeq	Spec. transformer	From concurrent programs to sequential programs by limiting context switches	[32, 126]
Liveness to reachability	Spec. transformer	From liveness specification to reachability specification	[30]
Memory-safety to reachability	Spec. transformer	From memory checking to reachability	[31, 81]
Specification decomposition	Spec. transformer	Partition into several smaller specifications	[83]
Testability transformation	Spec. transformer	Improve testability by transforming the program and coverage specification	[82]
ACSL2Witness	Witness transformer	Convert ACSL annotations to a software witness	[35]
Blast	Witness transformer	Convert a counterexample to a test case	[84]
Btor2-Cert	Witness transformer	Convert a software witness based on automata to a Btor2 witness	[33]
CPA-witness2test	Witness transformer	Convert a witness to a test case	[34]
CProver-witness2test	Witness transformer	Convert a witness to a test case	[34]
CPV	Witness transformer and Instrumentor	Convert a Btor2 hardware witness to an automata-based software witness via instrumentation	[29]
Witness testification	Witness transformer	Convert a witness to one with more information to replay the error path	[119]
Program slicing	Slicer	Slice a program while maintaining its functionalities	[85]
Symbiotic	Slicer	Apply slicing before symbolic execution	[65, 66]
Bubaak-Split	Splitter	Split a program and verify in parallel	[95]
Conditional data-flow analysis	Splitter	Structurally decompose CFA to accelerate data-flow analysis	[92]
LIV	Splitter and Annotator	Annotate a program and split into straight-line programs for witness validation	[24]
Program splitting	Splitter	Split a program into bottom and top	[89]

Approach/Tool	Type	Description	Ref.
Ranged analysis	Splitter	Split program paths into sets and run different analyses	[91]
Ranged symbolic execution	Splitter	Split program paths into sets to scale symbolic execution	[90]
VST-T	Splitter	Split a program into straight-line programs for verification	[94]
Conditional model checking	Pruner	Prune explored parts of a program based on witnesses	[22, 97]
Conditional testing	Pruner	Prune already-hit test goals based on test suites	[25]
Data-flow with transformation	Pruner	Simplify control-flow graph by propagating constant values	[96]
Cooperative automatic and interactive verifiers	Annotator	Exchange information between automatic and interactive tools via annotations	[35, 100]
Cooperative model checking and testing	Annotator	Annotate assumptions of model checkers to guide testers	[98, 99]
MetaVal	Annotator	Annotate a program with a witness and call verifiers for witness validation	[23]
CEGAR-PT	Reducer	Standalone framework for transforming loops in a program	[106]
Loop abstraction	Reducer	Abstract loop structures, e.g., by havocing variables in the loop	[104, 105]
Loop acceleration	Reducer	Replace loops with transitive closures to speed up convergence	[101, 102]
Loop shrinking	Reducer	Reduce the iterations of loops while maintaining overapproximation	[103]
Symbiotic for memory safety	Instrumentor	Add code to track allocated memory regions	[107]

Cooperative Verification. Cooperation between tools is a key to enable sharing the workload between components or tools. CoVeriTest [26, 124, 125] is an approach to combine two different approaches to program analysis, where the two components work on a shared data structure to explore the reachable abstract states. One component is a predicate analysis and the other is an explicit-value analysis, both having different strengths and performance characteristics. Cooperation leads to joining forces and obtaining an overall stronger analysis.

Conditional model checking [97, 127] is a technique to instruct a conditional model checker that parts described by a given input condition are to be considered correct and the model checker should verify only the state space that is outside the state space described by the input condition. The conditional model checker also produces an output condition, which describes the state space that it proved correct. The model checker does not make any claims about the state space

outside the condition, and leaves this (cooperatively) to another conditional model checker, which is in turn fed with the produced condition. Besides this sequential composition of conditional model checkers, the state space can also be split into two parts, one described by a condition and the other by its negation. Then, two conditional model checkers can run in parallel and verify the state space given to them in isolation, while contributing to a modular proof of correctness. The idea of conditional model checking was later also applied to testing [25]. Pruner-based conditional model checking [22] uses a transformation to prune a given C program according to a given condition to the state space that is not yet proven correct. Conditional model checking can be used to split a verification tasks into several smaller ones (see also [91]) and also to let each verifier verify the part of the program that matches its strengths.

Verifier Development. Already the first software model checker, Slam [128], was developed using the transformation approach, where the information exchange was done using Boolean programs. CEGAR was executed as follows: In a given CEGAR step, the component C2BP abstracted the C program to a Boolean program, using a given set of abstraction predicates (the precision). Then, the component Bebop [129] performed a model-checking pass on the Boolean program. If Bebop found a program path to the error, the path was checked for feasibility. If the path was feasible, a bug was found, if not, the infeasible path was further analyzed and the reason for infeasibility was used to refine the Boolean program, which was then given to the model checker. This approach has an easy-to-understand architecture, but the next decades set a high priority on performance tuning, for which it was easier to use a monolithic implementations in one tool, such as in Blast [130] or Cbmc [60]. We hope for a renaissance of transformation-based verification.

The software model checker for termination analysis Terminator [131, 132] also uses existing technology as components: Terminator uses the tool Rank-Finder [133] to generalize the counterexample, and then it refines the transition invariants with this generalization. For checking the validity of the transition invariants, Terminator reduces the problem to a reachability query, which can be solved by standard reachability verifiers. This work has shown that for tackling a problem as hard as analysis of software termination it is imperative to build on existing components, and that such an architecture can be successful.

C-CEGAR [21] demonstrates how to decompose counterexample-guided abstraction refinement (CEGAR) [128, 134] into its three major components, abstract-model exploration, feasibility check, and precision refinement. Instead of integrating those components inside one tool, the three components are implemented as three executable and independent tools, with a well-defined interface (in contrast to Slam, information is passed from component to component as verification witnesses [43], or using special-purpose file formats for paths and precisions). This construction from executable components makes it easier to substitute improved versions or alternatives for each of the three components.

CEGAR-based loop abstraction [105] is a technique to abstract and refine the control flow: in a coarse abstract model, loops are represented by their abstractions, and if the model turns out to be too imprecise, then the loops are refined, perhaps

by approximations, and on the most precise level by their original implementation. The tool CEGAR-PT [106] implements this approach as a sequence of program transformation, where the loop abstraction is applied to the original program, then the abstract program is given to a software verifier for exploration, then, if an infeasible counterexample is found, the program is transformed to a more refined program by using a more precise loop abstraction, and then given to the software verifier again, and so on. Implementing the loop abstraction as program transformation makes it possible to apply the loop abstraction as a preprocessing step to *any standard software verifier*. The transformation separates the concern of loop abstraction from the concern of program analysis.

Validator Development. After the first four validators for verification witnesses had been implemented, the first transformation-based witness validator was developed: MetaVal [23]. The idea was to reduce the problem of witness validation to the problem of reachability-safety analysis, by annotating the information from the witness into the program. The overall tool confirms a correctness (resp. violation) witness if the verifier can prove (resp. disprove) the safety of the program using the annotated information from the witness. This approach has the advantage that any safety verifier for C programs can be used as backend for MetaVal. This means, the development of witness validators can concentrate on the annotation, while the highly-tuned and already available verifiers can be used for verification.

LIV [24] goes one step further: It splits the input C program into straight-line programs [94], which are loop-free C programs, and those straight-line programs are handed off to a verification tool. While normal SMT-based verification tools compose verification conditions in a logic format, such as SMT-LIB 2 [12], LIV uses the standard programming language C to write the verification conditions in the form of straight-line programs. Since no loop is contained, even bounded model checkers can be used for verifying straight-line programs. Furthermore, this approach can be parallelized, because each straight-line program can be verified independently from the others.

Bridging Verification Communities. Formal verification is a large research area, and there are many different communities (according to the application domain) to which formal methods can be applied. It is understood in the research area that knowledge transfer between the communities is necessary. Such a transfer between communities can lead to unification and improvements of the state of the art. The transfer can occur in two forms, either the *knowledge* is transferred and applied to different domains (algorithms for hardware verification might be re-implemented for software verification, e.g., PDR [135], IMC [136], DAR [137]), or off-the-shelf *tools* are adapted to different domains.

Applying Software Verifiers to Hardware Problems. Btor2C [27] is a translator that takes as input a system written as hardware circuit in the language Btor2 [36] and produces as output a system written as a software program in the language C [10]. This makes it possible to use many verifiers for C programs (68 such verifiers are listed in an overview paper [138]), and experiments show that such new verifiers for hardware (that are based on transformation and software verification) can find

bugs in hardware circuits that state-of-the-art hardware verifiers such as ABC [56] and AVR [39] were not able to find. This creates a new problem: Witnesses [43, 139] (also known as certificates) for correctness or violation must be transformed back to refer to the original model, but there are solutions for this problem [33]. The transformation paradigm is a solution that makes it possible to apply advancements in software verification immediately to the hardware domain, without the need of transferring the implementations of the algorithms to the new domain.

Applying Hardware Verifiers to Software Problems. CPV is a software verifier for C programs that uses Btor2 as intermediate language and solves the given input verification problem by a two-step approach: First, it translates the input C program to a Btor2 circuit using Kratos2 [115]. Second, it uses the hardware model checkers ABC [56] and AVR [39] to solve the verification problem. Verification witnesses are transferred back to witnesses that refer to the original input C program. The hardware model checkers are configured to run several different algorithms, including IC3/PDR, interpolation-based model checking, and k-induction, in a portfolio manner. The transformation paradigm is a solution that makes it possible to apply advancements in hardware verification immediately to the software domain, again, without the need of transferring the implementations of the algorithms to the new domain.

Communicating via Verification Witnesses. Verification witnesses were originally introduced with the goal to justify the verification result and to be able to independently validate the result using the witness (with tools called witness checkers [44, 119, 140]). But it was quickly discovered that witnesses are much more important, enabling explainability and understandability by visualizing the content of the witnesses (e.g., supporting debugging [141]), and enabling communication between verification components (e.g., as explained above for C-CEGAR [21], or in CoVEGI [142]). All those approaches increase the overall usability of verification technology. In the following, we explain a few examples.

Witness to Test. Once a verification tool proves that the specification is violated, and a violation witness describes an error path through the program, it should be interesting to transform the violation witness to a test vector. The execution of the program on the test vector then exposes the specification violation at runtime. This approach of execution-based witness validation [34] is implemented in the tools CPA-witness2test (based on the CPAchecker verification framework) and CProver-witness2test (based on the CProver verification framework). This approach makes it possible to bridge the gap between verification and testing. Testing is already well integrated into development workflows, and the results of verification runs that resulted in discovered bugs can be integrated easily into the existing testing workflow by adding the generated test vectors to the test suite.

Translating Software Witnesses to Hardware Witnesses. Witnesses are also important to communicate the justification of verification results from one language for systems to others. A witness always refers to the model (e.g., to values or relations of the variables used in the model). For example, Btor2-Cert [33] has a component that translates the software witness produced by a software

verifier (referring to the C program that the verifier analyzed) to a hardware witness (referring to the original Btor2 circuit). In the opposite direction, the tool CPV [29] must deliver software witnesses, but uses hardware model checkers as backend, and thus, needs to translate hardware witnesses to software witnesses, such that the resulting witness refers to the input C program.

Cooperation of Automatic and Interactive Verifiers. Verification witnesses also open up many opportunities to connect automatic and interactive verifiers. For example, automatic verifiers write their invariants into witnesses, while interactive verifiers ask the users to insert their invariants as annotations into the program. Interactive verifiers can benefit from invariants that were generated from automatic verifiers. Researchers developed annotators [35] that, for example, take as input a C program and a correctness witness (with invariants) and produce as output a C program with ACSL [120] annotations. This annotated program can be given as input to the interactive verifier Frama-C [121]. The experiments of the study [35] report that Frama-C can verify more programs with the help of the automatically produced annotations. This translation makes it possible to use any automatic verifier to produce invariants (or contracts) in order to help the users of an interactive verifier with information that can be derived automatically.

6 Conclusion

Formal verification is a hard problem and we need to leverage all possible approaches to solve this important problem. We advocate the approach of *modular transformation*, which uses standalone components for transformation in order to apply verification technology. We introduce the necessary notions and survey several approaches and tools that use transformation, in order to illustrate that this paradigm can be useful. We make a case for *verification by transformation* by explaining some applications of transformation to construct modular tools for formal verification and give hints on their benefits and impact. We hope that readers find this view on formal verification inspiring and either contribute more transformations or use some of the existing transformations.

Data-Availability Statement. We have added the transformers used by the tools in the FM-Tools Repository to their respective meta data under key `used_actors`. A generated web site is available at: `https://fm-tools.sosy-lab.org/`

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 536040111 (Bridge) and 378803395 (ConVeY), as well as by the LMU PostDoc Support Fund.

Acknowledgement. We would like to thank Joost-Pieter Katoen for his significant contributions to research tools, including his own tools and his support of tool development in the community. For example, he was involved in establishing the ETAPS Test-of-Time Tool Award to emphasize the importance of reliable and well-maintained research tools. We also thank the reviewers of our article for their constructive comments on how to extend our work on this.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT (2008), https://www.worldcat.org/isbn/978-0-262-02649-9
2. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Proc. IFM. pp. 1–20. LNCS 2999, Springer (2004). https://doi.org/10.1007/978-3-540-24756-2_1
3. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
4. 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
5. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., 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
6. Li, X., Li, X., Dall, C., Gu, R., Nieh, J., Sait, Y., Stockwell, G.: Design and verification of the Arm confidential compute architecture. In: Proc. OSDI. pp. 465–484. USENIX Association (2022), https://www.usenix.org/system/files/osdi22-li.pdf
7. Fox, A.C.J., Stockwell, G., Xiong, S., Becker, H., Mulligan, D.P., Petri, G., Chong, N.: A verification methodology for the Arm confidential computing architecture: From a secure specification to safe implementations. Proc. ACM Program. Lang. **7**(OOPSLA1), 376–405 (2023). https://doi.org/10.1145/3586040
8. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS (2009)
9. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_11
10. Kernighan, B., Ritchie, D.: The C Programming Language. Prentice Hall (1978)
11. ISO/IEC JTC 1/SC 22: ISO/IEC 9899-2018: Information technology — Programming Languages — C. International Organization for Standardization (2018), https://www.iso.org/standard/74528.html
12. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., University of Iowa (2010), https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf
13. 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
14. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
15. DeMarco, T.: Structured Analysis and System Specification. Prentice Hall, facsimile edn. (1979)
16. Szyperski, C.A., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 2nd edn. (2002)
17. Ritchie, D., Thompson, K.: The UNIX time-sharing system. Commun. ACM **17**(7), 365–375 (1974). https://doi.org/10.1145/361011.361061
18. Raymond, E.S.: The Art of UNIX Programming. Addison-Wesley, 1st edn. (2003)

19. Lattner, C., Adve, V.S.: Llvm: A compilation framework for lifelong program analysis and transformation. In: Proc. CGO. pp. 75–88. IEEE (2004). https://doi.org/10.1109/CGO.2004.1281665
20. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: MLIR: Scaling compiler infrastructure for domain-specific computation. In: Proc. CGO. pp. 2–14. IEEE (2021). https://doi.org/10.1109/CGO51591.2021.9370308
21. Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing software verification into off-the-shelf components: An application to CEGAR. In: Proc. ICSE. pp. 536–548. ACM (2022). https://doi.org/10.1145/3510003.3510064
22. 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
23. 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
24. Beyer, D., Spiessl, M.: LIV: A loop-invariant validation using straight-line programs. In: Proc. ASE. pp. 2074–2077. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00214
25. 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
26. Beyer, D., Jakobs, M.C.: Cooperative verifier-based testing with CoVeriTest. Int. J. Softw. Tools Technol. Transfer **23**(3), 313–333 (2021). https://doi.org/10.1007/s10009-020-00587-8
27. Beyer, D., Chien, P.C., Lee, N.Z.: Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator. In: Proc. TACAS (2). pp. 152–172. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_12
28. Tafese, J., Garcia-Contreras, I., Gurfinkel, A.: Btor2MLIR: A format and toolchain for hardware verification. In: Proc. FMCAD. pp. 55–63. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_13
29. 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
30. 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
31. 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
32. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods Syst. Des. **35**(1), 73–97 (2009). https://doi.org/10.1007/S10703-009-0078-9
33. Ádám, Z., Beyer, D., Chien, P.C., Lee, N.Z., Sirrenberg, N.: Btor2-Cert: A certifying hardware-verification framework using software analyzers. In: Proc. TACAS (3). pp. 129–149. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_7
34. 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

35. Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: Proc. SEFM. p. 111–128. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_7
36. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. CAV. pp. 587–595. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
37. 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
38. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV. pp. 343–361. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
39. Goel, A., Sakallah, K.: AVR: Abstractly verifying reachability. In: Proc. TACAS. pp. 413–422. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_23
40. 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
41. 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
42. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Tech. Rep. 07/1, Institute for Formal Models and Verification, Johannes Kepler University (2007). https://doi.org/10.35011/fmvtr.2007-1
43. 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
44. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011). https://doi.org/10.1016/j.cosrev.2010.09.009
45. Besson, F., Jensen, T.P., Pichardie, D.: Proof-carrying code from certified abstract interpretation and fixpoint compression. TCS **364**(3), 273–291 (2006). https://doi.org/10.1016/j.tcs.2006.08.012
46. Johannsen, C., Nukala, K., Dureja, R., Irfan, A., Shankar, N., Tinelli, C., Vardi, M.Y., Rozier, K.Y.: The MoXI model exchange tool suite. In: Proc. CAV. pp. 203–218. LNCS 14681, Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_10
47. IEEE standard for Verilog hardware description language (2006). https://doi.org/10.1109/IEEESTD.2006.99495
48. Snyder, W.: Verilator. https://www.veripool.org/verilator/, accessed: 2023-01-29
49. Smrcka, A., Vojnar, T.: Verifying parametrised hardware designs via counter automata. In: Proc. HVC. pp. 51–68. LNCS 4899, Springer (2007). https://doi.org/10.1007/978-3-540-77966-7_8
50. IEEE standard for VHDL language reference manual (2019). https://doi.org/10.1109/IEEESTD.2019.8938196
51. McMillan, K.L.: Symbolic Model Checking. Springer (1993). https://doi.org/10.1007/978-1-4615-3190-6
52. Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: Verilog2SMV: A tool for word-level verification. In: Proc. DATE. pp. 1156–1159 (2016), https://ieeexplore.ieee.org/document/7459485

53. Minhas, M., Hasan, O., Saghar, K.: Ver2Smv: A tool for automatic Verilog to SMV translation for verifying digital circuits. In: Proc. ICEET. pp. 1–5 (2018). https://doi.org/10.1109/ICEET1.2018.8338617
54. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Proc. CAV. pp. 334–342. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
55. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Source-code repository of Btor2, BtorMC, and Boolector 3.0. https://github.com/Boolector/btor2tools, accessed: 2023-01-29
56. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proc. CAV. pp. 24–40. LNCS 6174, Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_5
57. The CIRCT project: Circuit IR compilers and tools. https://circt.llvm.org/, accessed: 2024-05-14
58. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proc. PLDI. pp. 203–213. ACM (2001). https://doi.org/10.1145/378795.378846
59. 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
60. 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
61. 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
62. Falke, S., Merz, F., Sinz, C.: The bounded model checker Llbmc. In: Proc. ASE. pp. 706–709. IEEE (2013). https://doi.org/10.1109/ASE.2013.6693138
63. Priya, S., Su, Y., Bao, Y., Zhou, X., Vizel, Y., Gurfinkel, A.: Bounded model checking for Llvm. In: Proc. FMCAD. pp. 214–224. IEEE (2022). https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_28
64. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
65. 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
66. 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
67. Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementations. In: Proc. CAV. pp. 106–113. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_7
68. DeLine, R., Leino, R.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005)
69. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: Proc. FMCO. pp. 115–137. LNCS 4111, Springer (2005). https://doi.org/10.1007/11804192_6

70. Bjørner, N.S., McMillan, K.L., Rybalchenko, A.: Program verification as satisfiability modulo theories. In: Proc. SMT. EPiC Series in Computing, vol. 20, pp. 3–11. EasyChair (2012). https://doi.org/10.29007/1l7f
71. Johnson, D.S., Trick, M.A. (eds.): Cliques, Coloring, and Satisfiability, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26. DIMACS/AMS (1996). https://doi.org/10.1090/DIMACS/026
72. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: Proc. CAV. pp. 846–862. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_59
73. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained Horn clauses using syntax and data. In: Proc. FMCAD. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603011
74. Hojjat, H., Rümmer, P.: The ELDARICA Horn solver. In: Proc. FMCAD. pp. 1–7. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603013
75. Blicha, M., Britikov, K., Sharygina, N.: The Golem Horn solver. In: Proc. CAV. pp. 209–223. LNCS 13965, Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_10
76. Kroening, D., Purandare, M.: EBMC. http://www.cprover.org/ebmc/, accessed: 2023-01-29
77. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Proc. VMCAI. pp. 41–62. LNCS 9583, Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2
78. Dellacherie, S., Devulder, S., Lambert, J.L.: Software verification based on linear programming. In: Proc. FM. pp. 1147–1165. LNCS 1709, Springer (1999). https://doi.org/10.1007/3-540-48118-4_11
79. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proc. CAV. pp. 3–29. LNCS 10426, Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_1
80. Zhang, Y., Zhao, Z., Chen, G., Song, F., Zhang, M., Chen, T., Sun, J.: QVIP: An ILP-based formal verification approach for quantized neural networks. In: Proc. ASE. pp. 82:1–82:13. ACM (2022). https://doi.org/10.1145/3551349.3556916
81. 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
82. 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
83. Apel, S., Beyer, D., Mordan, V.O., Mutilin, V.S., Stahlbauer, A.: On-the-fly decomposition of specifications in software model checking. In: Proc. FSE. pp. 349–361. ACM (2016). https://doi.org/10.1145/2950290.2950349
84. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
85. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984). https://doi.org/10.1109/tse.1984.5010248
86. Chalupa, M., Strejček, J.: Evaluation of program slicing in software verification. In: Proc. IFM. pp. 101–119. LNCS 11918, Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_6
87. Westhofen, L., Berger, P., Katoen, J.P.: Benchmarking software model checkers on automotive code. In: Proc. NFM. pp. 133–150. LNCS 12229, Springer (2020). https://doi.org/10.1007/978-3-030-55754-6_8

88. Berger, P.: Applying Software Model Checking: Experiences and Advancements. Ph.D. thesis, RWTH Aachen (2023)
89. Lifschitz, V., Turner, H.: Splitting a logic program. In: Proc. ICLP. pp. 23–37. MIT Press (1994). https://doi.org/10.7551/mitpress/4316.003.0014
90. Siddiqui, J.H., Khurshid, S.: Scaling symbolic execution using ranged analysis. In: Leavens, G.T., Dwyer, M.B. (eds.) Proc. SPLASH. pp. 523–536. ACM (2012). https://doi.org/10.1145/2384616.2384654
91. Haltermann, J., Jakobs, M.C., Richter, C., Wehrheim, H.: Parallel program analysis via range splitting. In: Proc. FASE. pp. 195–219 (2023). https://doi.org/10.1007/978-3-031-30826-0_11
92. Sherman, E., Dwyer, M.B.: Structurally defined conditional data-flow static analysis. In: Proc. TACAS (2). pp. 249–265. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_15
93. Beyer, D., Friedberger, K.: Domain-independent multi-threaded software model checking. In: Proc. ASE. pp. 634–644. ACM (2018). https://doi.org/10.1145/3238147.3238195
94. Zhou, L.: Foundationally sound annotation verifier via control flow splitting. In: Proc. SPLASH. pp. 69–71. ACM (2022). https://doi.org/10.1145/3563768.3563956
95. Chalupa, M., Richter, C.: Bubaak-SpLit: Split what you cannot verify (competition contribution). In: Proc. TACAS (3). pp. 353–358. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_20
96. Lerner, S., Grove, D., Chambers, C.: Composing data-flow analyses and transformations. In: Proc. POPL. pp. 270–282. ACM (2002). https://doi.org/10.1145/503272.503298
97. 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
98. 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
99. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proc. ICSE. pp. 144–155. ACM (2016). https://doi.org/10.1145/2884781.2884843
100. Shankar, N.: Combining model checking and deduction. In: Handbook of Model Checking., pp. 651–684. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_20
101. Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces. Ph.D. thesis, Faculté des Sciences Appliquées de Université de Liège (1998)
102. Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: Proc. ATVA. pp. 474–488. LNCS 3707, Springer (2005). https://doi.org/10.1007/11562948_35
103. Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property checking array programs using loop shrinking. In: Proc. TACAS (1). pp. 213–231. LNCS 10805, Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_12
104. 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
105. 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

106. 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
107. 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
108. Beyer, D., Kanav, S., Richter, C.: Construction of verifier combinations based on off-the-shelf verifiers. In: Proc. FASE. pp. 49–70. Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_3
109. Haltermann, J., Wehrheim, H.: Exchanging information in cooperative software validation. Softw. Syst. Model. **23**(3), 695–719 (2024). https://doi.org/10.1007/S10270-024-01155-3
110. Mukherjee, R., Tautschnig, M., Kroening, D.: v2c: A Verilog to C translator. In: Proc. TACAS. pp. 580–586. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_38
111. Biere, A., van Dijk, T., Heljanko, K.: Hardware model-checking competition 2017. In: Proc. FMCAD. p. 9. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102233
112. Biere, A., Froleyks, N., Preiner, M.: 11th hardware model-checking competition (HWMCC 2020). http://fmv.jku.at/hwmcc20/, accessed: 2023-01-29
113. 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
114. Long, J.: Reasoning about High-Level Constructs in Hardware/Software Formal Verification. Ph.D. thesis, EECS Department, University of California, Berkeley (2017), http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-150.html
115. 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
116. Cimatti, A., Griggio, A., Tonetta, S.: The VMT-LIB language and tools. In: Proc. SMT. CEUR Workshop Proceedings, vol. 3185, pp. 80–89. CEUR-WS.org (2022)
117. Rozier, K.Y., Dureja, R., Irfan, A., Johannsen, C., Nukala, K., Shankar, N., Tinelli, C., Vardi, M.Y.: MoXI: An intermediate language for symbolic model checking. In: Proc. SPIN. LNCS , Springer (2024)
118. Beyer, D., Chien, P.C., Lee, N.Z.: MoXIchecker: An extensible model checker for MoXI. arXiv/CoRR **2407**(15551) (July 2024). https://doi.org/10.48550/arXiv.2407.15551
119. 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
120. 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
121. 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
122. Beyer, D.: Conservation and accessibility of tools for formal methods. In: Proc. Festschrift Podelski 65th Birthday. Springer (2024)
123. Beyer, D.: Tools for formal methods. https://fm-tools.sosy-lab.org/, accessed: 2024-08-21

124. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
125. Jakobs, M.C., Richter, C.: CoVeriTest with adaptive time scheduling (competition contribution). In: Proc. FASE. pp. 358–362. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_18
126. 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
127. 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
128. Ball, T., Rajamani, S.K.: Boolean programs: A model and process for software analysis. Tech. Rep. MSR Tech. Rep. 2000-14, Microsoft Research (2000), https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2000-14.pdf
129. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: Proc. SPIN. pp. 113–130. LNCS 1885, Springer (2000). https://doi.org/10.1007/10722468_7
130. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer **9**(5-6), 505–525 (2007). https://doi.org/10.1007/s10009-007-0044-z
131. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Proc. CAV. pp. 415–418. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_37
132. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proc. PLDI. pp. 415–426. ACM (2006). https://doi.org/10.1145/1133981.1134029
133. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Proc. VMCAI. pp. 239–251. LNCS 2937, Springer (2004). https://doi.org/10.1007/978-3-540-24622-0_20
134. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
135. Lange, T., Neuhäußer, M.R., Noll, T., Katoen, J.P.: IC3 software model checking. Int. J. Softw. Tools Technol. Transf. **22**(2), 135–161 (2020). https://doi.org/10.1007/S10009-019-00547-X
136. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. J. Autom. Reasoning (2024). https://doi.org/10.1007/s10817-024-09702-9, preprint: https://doi.org/10.48550/arXiv.2208.05046
137. Beyer, D., Chien, P.C., Jankola, M., Lee, N.Z.: A transferability study of interpolation-based hardware model checking for software verification. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3660797
138. Beyer, D., Podelski, A.: Software model checking: 20 years and beyond. In: Principles of Systems Design. pp. 554–582. LNCS 13660, Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_27
139. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. Springer (2024)
140. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **1610**(06229) (October 2016)

141. Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: Proc. CAV (2). pp. 502–509. LNCS 9780, Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_28
142. Haltermann, J., Wehrheim, H.: CoVEGI: Cooperative verification via externally generated invariants. In: Proc. FASE. pp. 108–129. LNCS 12649 (2021). https://doi.org/10.1007/978-3-030-71500-7_6

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.