Btor2-Cert: A Certifying Hardware-Verification Framework Using Software Analyzers

Zsófia Ádám[1,2], Dirk Beyer[2], Po-Chun Chien[2], Nian-Ze Lee[2], and Nils Sirrenberg[2]

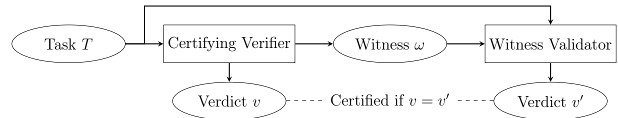
[1]Department of Measurement and Information Systems, Budapest University of Technology and Economics, Hungary
[2]Department of Computer Science, LMU Munich, Germany

Abstract. Formal verification is essential but challenging: Even the best verifiers may produce wrong verification verdicts. *Certifying* verifiers enhance the confidence in verification results by generating a *witness* for other tools to validate the verdict independently. Recently, translating the hardware-modeling language BTOR2 to software, such as the programming language C or LLVM intermediate representation, has been actively studied and facilitated verifying hardware designs by software analyzers. However, it remained unknown whether witnesses produced by software verifiers contain helpful information about the original circuits and how such information can aid hardware analysis. We propose a certifying and validating framework BTOR2-CERT to verify safety properties of BTOR2 circuits, combining BTOR2-to-C translation, software verifiers, and a new witness validator BTOR2-VAL, to answer the above open questions. BTOR2-CERT translates a software *violation witness* to a BTOR2 violation witness; As the BTOR2 language lacks a format for *correctness witnesses*, we encode invariants in software correctness witnesses as BTOR2 circuits. The validator BTOR2-VAL checks violation witnesses by circuit simulation and correctness witnesses by *validation via verification*. In our evaluation, BTOR2-CERT successfully utilized software witnesses to improve quality assurance of hardware. By invoking the software verifier CBMC on translated programs, it uniquely solved, with confirmed witnesses, 8 % of the unsafe tasks for which the hardware verifier ABC failed to detect bugs.

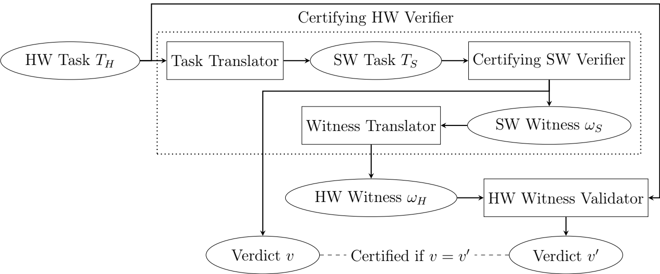
Keywords: Hardware verification · Software verification · Verification witnesses · Witness validation · Word-level circuit · BTOR2 · SMT · SAT

1 Introduction

Certifying algorithms [1] generate a *certificate* alongside the computed solution such that *proof checkers* can independently validate the solution to increase users' trust and the explainability of the results. In the model-checking community, a certificate to explain a verdict for a verification task is called a *witness* [2], and verifiers able to generate witnesses are called *certifying verifiers*. Witnesses can be independently checked by *witness validators* to confirm the verification



(a) Certifying and validating model checking



(b) Translating SW witnesses to HW for validation

Fig. 1: A certifying hardware-verification framework using software analyzers

results. Figure 1a shows a generic workflow for *certifying and validating model checking*. After a certifying verifier produces a verdict v and a witness ω on a task T, a witness validator takes T and ω as input and checks if the information in ω is enough to reestablish the results of the verifier on T. The outcome of the verifier is *certified* if its verdict v and the validator's verdict v' are consistent. In the rest of the paper, we use *certifying model checking* interchangeably with *certifying and validating model checking* when it is clear from the context that a framework contains both a certifying verifier and a witness validator. For reachability properties, if a model violates a safety specification, a *violation witness* [3] may contain external inputs to the model to replay the erroneous execution trace. If the safety specification is satisfied, a *correctness witness* [4] could record invariants of the model to reconstruct a safety proof. Section 2 presents a brief survey on witness validation in the formal-methods community.

Recently, hardware-to-software translators [5, 6] from the hardware-modeling language Btor2 [7], a prevailing format for word-level hardware model checking used in the Hardware Model Checking Competitions (HWMCC) [8, 9], have been proposed to facilitate the application of software analyzers to hardware circuits. Tools Btor2C [5] and Btor2MLIR [6] translate Btor2 circuits to behaviorally equivalent imperative software in the programming language C [10] and the intermediate representation used by the compilation toolchain LLVM [11], respectively, and enable any software analyzer for C or LLVM-bytecode programs to inspect Btor2 circuits. In an experiment on more than 1 000 Btor2 circuits [5], software verifiers for C programs are shown to detect more bugs than the best hardware model checkers by preprocessing the original circuit with Btor2C and analyzing the translated C program. However, in this previous work [5], only the verdicts of software verifiers but not the witnesses, which contain the information and reasoning behind a verdict, are transferred back to the hardware domain. In other words, the results of software verifiers on Btor2 circuits are

not certified, and hence hardware designers may not trust software verifiers for analyzing their circuits.

1.1 Our Motivation and Contributions

Motivated to mitigate the aforementioned threat to reliability and leverage the capability of software verifiers to generate witnesses, we investigate the following open questions in this work: (1) *whether the software witnesses for translated programs contain useful information about original circuits* and (2) *how to employ the information to aid hardware quality assurance*. Our contributions are summarized below.

A Certifying Framework for HW Verification with SW Analyzers. Figure 1b shows the proposed certifying and validating hardware-verification framework based on software analyzers to approach the open questions. The framework translates a hardware-verification task T_H to a software task T_S and applies software verifiers to T_S. After obtaining a software witness ω_S, it encodes relevant information from ω_S in the form of a hardware witness ω_H and validates the verdict returned by software verifiers with ω_H. We instantiate the framework in a tool Btor2-Cert for verifying Btor2 circuits with certified verdicts. In addition to preprocessing Btor2 circuits with Btor2C [5] and invoking model checkers for the translated C programs, such as CPAchecker [12], Cbmc [13], Esbmc [14], and UAutomizer [15], Btor2-Cert features a **translator** from software witnesses to Btor2 witnesses and a **witness validator** Btor2-Val to check Btor2 witnesses. Section 4 shows our tool architecture.

Note that the framework in Fig. 1b is not limited to Btor2C and verifiers for C programs. For example, one could also materialize the concept with the translator Btor2MLIR [6], analyzers for LLVM-bytecode programs [11], such as Klee [16], Smack [17], and SeaHorn [18], and a corresponding LLVM-to-Btor2 witness translator. There also exist translators [19, 20, 21] from Verilog [22] circuits to C programs or SMV [23] models. We choose Btor2C for task translation because many verifiers for C programs participating in the International Competitions on Software Verification (SV-COMP) [24] can generate witnesses in a standardized and exchangeable format [2].

A Translator from Software Witnesses to Btor2 Witnesses. Btor2-Cert translates software violation witnesses in the format used in SV-COMP [24] to the format defined by the Btor2 language [7]. For tasks satisfying their specifications, as there is no native format for correctness witnesses in Btor2, Btor2-Cert extracts the invariants in software witnesses and represents them as Btor2 circuits, whose inputs refer to the state variables of the original circuit. The advantages of not inventing a new format but reusing the existing Btor2 language are twofold: First, Btor2 extends SMT-LIB 2 [25] and provides the required operations on the word level to accommodate most invariants derived by software verifiers. Second, Btor2 is supported by many hardware model checkers participating in HWMCC [8, 9] and offers a suite Btor2Tools [26] of utility tools for parsing and simulation, which simplifies further development around the Btor2 format.

A Validator for Btor2 Witnesses. To validate the witnesses for Btor2 circuits, we develop Btor2-Val, a portfolio-based witness validator involving hardware simulators and verifiers. Btor2-Val validates violation witnesses by invoking the simulator BtorSim from Btor2Tools [26]. For correctness witnesses, Btor2-Val follows the *validation-via-verification* approach [27] by instrumenting the original Btor2 circuit with the circuit representing the invariant and verifying the instrumented circuit. The instrumented circuit satisfies the modified safety property if the invariant can be used to reconstruct the proof of correctness. Hardware verifiers are employed to check the instrumented circuits. Btor2-Val leverages CoVeriTeam [28], a framework for cooperative verification, to coordinate the underlying hardware simulators and verifiers.

Enhancing Confidence in SW Verifiers on HW Designs. We evaluate Btor2-Cert on more than 1 000 Btor2 circuits to study its capability of providing certified verification results using software analyzers. In the experiment,

- the witness translator was able to translate every violation witness and 97 % of the correctness witnesses produced by software verifiers,
- the combination of witness translation and Btor2-Val outperformed mature software witness validators in both effectiveness and efficiency, and
- Btor2-Cert provided **certified** results computed by software verifiers on some Btor2 circuits that the best hardware model checkers failed to verify.

The conceptual message conveyed by Btor2-Cert is *software analyzers can derive useful information about circuits and complement conventional hardware model checkers with trustworthy results*. Our contributions have a positive impact on analyzing hardware designs with software verifiers. The proposed framework Btor2-Cert is open-source and available online (more information in Sect. 4).

2 Related Work

Generating and validating witnesses for analysis results have been studied throughout the entire verification toolchain from satisfiability solvers to model checkers. In the following, we briefly review witness validation and compare our work to a recent certifying verification framework [29, 30, 31] targeting k-induction [32].

2.1 Witness Validation

For satisfiability solving, the competitions on propositional SAT solvers [33, 34] use the DRAT format [35] to encode the certificates of unsatisfiability and independent validators [36, 37] to check the proofs. The competitions on SMT solving verify models to satisfiable formulas with the tool Dolmen [38]. Certifications for quantified Boolean formulas have also been investigated [39, 40].

For model checking, an early work [41] suggests generating a deductive proof from the run of model checkers with extra bookkeeping steps. In HWMCC [8, 9], the Btor2 [7] language defines a format for violation witnesses as a sequence of input values and initial values for registers that lead to an erroneous execution. However, Btor2 has no format for correctness witnesses. The competitions on

automated termination analysis [42] use the format CPF [43], and in SV-COMP [24], a GraphML-based format [2] is used to describe software witnesses as automata. In addition to the properties commonly used in tool competitions, a recent work extends proof generation of model checking to full LTL properties [44].

Numerous approaches have been invented for validating software witnesses. Methods to validate correctness witnesses include a parallel extension [45] of k-induction, program instrumentation with invariants and re-verification [27] (referred to as *validation via verification* in the publication), and program decomposition into several straight-line sub-programs [46]. *Execution-based validation* [47] is an elegant approach to validate violation witnesses. It extracts a sequence of external input values from a violation witness and employs debuggers or simulators to testify the reachability of an error location. Our witness validator Btor2-Val leverages validation via verification and execution-based validation. More details are given in Sect. 5 and Sect. 6, respectively. In our evaluation, the proposed validator Btor2-Val (together with the witness translator) competed well against the winners in the witness-validation track of SV-COMP 2023 [24].

2.2 Validating k-Inductiveness of Properties in Hardware Models

Given a sequential circuit and a number k as input, the tool Certifaiger [29, 30] aims to validate that the safety property of the input circuit is k-inductive. Composing a k-induction-based hardware model checker and Certifaiger yields a certifying and validating model checker (as depicted in Fig. 1a), whose witnesses are the inductive length k. The key differences between the proposed framework in Fig. 1b and this framework [29, 30] for k-inductiveness are as follows.

First, our validator Btor2-Val expects a candidate invariant in the correctness witness but does not restrict the algorithms used by software verifiers. In contrast, Certifaiger expects a candidate inductive length k and thus can only validate results of k-induction-based model checkers. Second, to validate witnesses, Btor2-Val relies on validation via verification [27] and invokes model checkers because the candidate invariant may not be inductive. In comparison, Certifaiger avoids model checking and reduces the validation problem to several SAT checks since it assumes the safety property to be k-inductive. To sum up, our framework complements the existing work [29, 30] by considering candidate invariants as witnesses. Its applicability to algorithms other than k-induction comes at the expense of potentially more complex validation procedure. Certifaiger is further extended to accommodate temporal decomposition [48] as preprocessing to simplify the verification tasks [31], which has not yet been considered in our framework and is an important direction of future work.

3 Background

To facilitate the discussion in the rest of this manuscript, we provide prerequisite knowledge on model checking and witness validation from the literature.

A state-transition system $\mathcal{M}$ is described by two predicates $I(s)$ and $TR(s, s')$ over states s and s' of $\mathcal{M}$, which encode the *initial states* and *transition relation* ($TR(s, s')$ is true if s can transit to s' via one step) of $\mathcal{M}$, respectively. An

```
1 sort bitvec 8
2 sort bitvec 1
3 constd 1 42
4 constd 1 2
5 zero 1
6 state 1 ; a
7 state 1 ; b
8 input 1 ; in
9 init 1 6 4 ; a init to 2
10 init 1 7 5 ; b init to 0
11 eq 2 6 5 ; a == 0
12 eq 2 7 4 ; b == 2
13 eq 2 8 3 ; in == 42
14 and 2 11 12
15 and 2 13 14
16 bad 15
17 one 1
18 srl 1 6 17
19 xor 1 7 17
20 next 1 6 18
21 next 1 7 19
```

(a) Btor2 circuit

```
1  extern void abort(void);
2  extern unsigned char nondet_uchar();
3  void main() {
4    typedef unsigned char SORT_1;
5    SORT_1 a = nondet_uchar();
6    SORT_1 b = nondet_uchar();
7    a = 2;
8    b = 0;
9    for (;;) {
10     SORT_1 in = nondet_uchar();
11     if (a == 0 && b == 2 && in == 42) {
12       ERROR: abort();
13     }
14     a = a >> 1;
15     b = b ^ 1;
16   }
17 }
```

(b) C program (simplified for demo)

Fig. 2: An example Btor2 circuit and its translated C program

invariant $Inv(s)$ of a system $\mathcal{M}$ is a predicate over states of $\mathcal{M}$ such that $Inv(s)$ *is true for every reachable state* s *of* $\mathcal{M}$. We denote "Inv is an invariant of $\mathcal{M}$" by $\mathcal{M} \models Inv$. A *safety-verification task* consists of a state-transition system $\mathcal{M}$ and a safety property $P(s)$. We say a safety-verification task (or a verification task for short) is *safe* if $\mathcal{M} \models P$ and *unsafe* otherwise. Given a verification task of $\mathcal{M}$ and P, the problem of *model checking* asks whether $\mathcal{M} \models P$ or not. In practice, state-transition systems manifest themselves as sequential digital circuits or programs. In the following, we briefly introduce the modeling languages used in HWMCC [8, 9] and SV-COMP [24] with a running example.

3.1 The Btor2 Language for Word-Level Circuits

The Btor2 hardware-modeling language [7] was invented to describe model-checking problems of word-level sequential circuits. It extends the bit-level AIGER format [49] with data sorts of bit-vectors and arrays and inherits word-level operations from SMT-LIB 2 [25]. Figure 2a shows an example Btor2 circuit. The circuit has two state variables `a` and `b` and an external input `in`, defined in lines `6`-`8`, respectively. The states and input are bit-vectors of width 8 (the sort `bitvec 8` defined in line `1`). Variables `a` and `b` are initialized to 2 and 0, respectively. In each iteration, variable `a` is right-shifted by 1 bit (line `18`), and variable `b` is bitwise XOR-ed with 1 (line `19`). Indicated by the keyword `bad` in line `16`, a property violation happens if variable `a` equals 0, variable `b` equals 2, and input `in` equals 42. The example Btor2 circuit satisfies its safety property because variable `b` never equals 2. However, if variable `b` is initialized to a different value at line `10` (marked in red), say 2, a property violation will be triggered after two steps of state transition if 42 is given as the external input in the last iteration.

Translating Btor2 Circuits to C Programs. Btor2C [5] is a lightweight translator from the Btor2 language to the programming language C [10]. It

encodes Btor2 data sorts with unsigned integers and static arrays, expresses Btor2 operations with corresponding operators of C, and uses an infinite loop to model the execution of a sequential circuit. Given the example Btor2 circuit in Fig. 2a as input, Btor2C generates a translated C program[1] shown in Fig. 2b. Btor2C follows the rules of SV-COMP [24] to encode safety-verification tasks for C programs, so *compositional* hardware model checkers for Btor2 circuits can be readily formed by combining software verifiers participating in SV-COMP as verification engines and Btor2C as preprocessing. In an extensive experiment [5], software verifiers are shown to detect more bugs in Btor2 circuits than the best *conventional* hardware model checkers, such as ABC [50] and AVR [51].

3.2 Representing Software Witnesses as Automata

Software witnesses can be represented as *protocol automata* [2], describing program invariants needed to construct a safety proof or program paths leading to a property violation. A letter in the alphabet of such a protocol automaton is a pair of a set of program edges and a condition over program variables. The set of program edges indicates the control flow, and the condition can be used to restrict the state space of the program. Program invariants that should hold at a certain program location can be annotated to a protocol automaton. In the following, we give an example correctness witness for the C program in Fig. 2b and an example violation witness for the same C program but with line `8` commented out.

Correctness Witnesses. Figure 3 shows an example correctness witness for the C program in Fig. 2b. The correctness witness shows that a program invariant `b>=0 && b<=1` is established once line `8` is executed. Indeed, variable `b` switches between 0 and 1 after being initialized, and `b>=0 && b<=1` is an invariant at the loop head of the program. A program invariant is stored as a C expression in a software witness and hence potentially more compact than invariants represented in other formalisms, e.g., a bit-level AIGER [49] circuit.

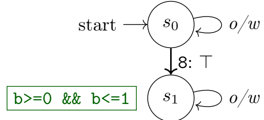


Fig. 3: A correctness witness

Violation Witnesses. Figure 4 shows an example violation witness for the modified C program with variable `b` uninitialized (by commenting out line `8` in Fig. 2b). The violation witness shows how to reach the error in line `12` of the C program. First, it assumes the value of variable `b` to be 2 via the condition when line `6` is executed. Second, it goes to the next state when line `10` is executed for the first two times. Third, it assumes the external input to be 42 when line `10` is executed for the third time. Indeed, the error in line `12` can be reached if variable `b` gets an initial value of 2 and the external input equals 42 in the third loop iteration.

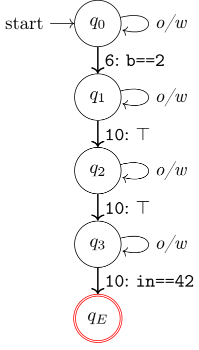
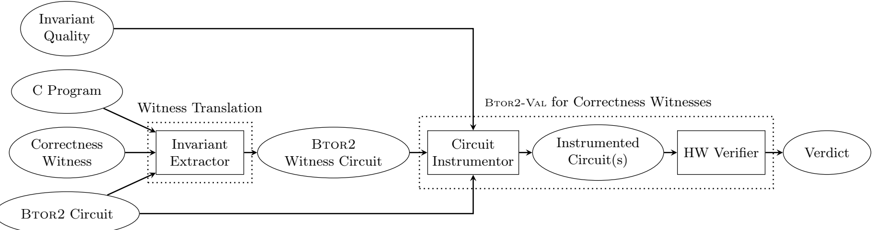
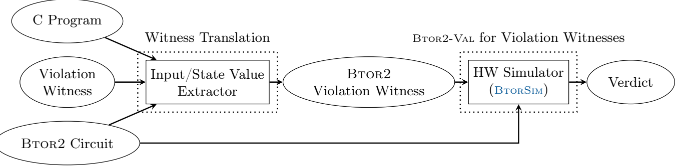


Fig. 4: A violation witness

[1] The intermediate variables in the actual output program of Btor2C are omitted.



(a) Validating correctness witnesses by circuit instrumentation and verification



(b) Validating violation witnesses by circuit simulation

Fig. 5: Witness translation and validation in Btor2-Cert and Btor2-Val

4 Architecture of Btor2-Cert and Btor2-Val

We instantiate the proposed certifying and validating hardware-verification framework in Fig. 1b as Btor2-Cert[2] with the Btor2-to-C translator Btor2C [5], model checkers for C programs [52] that can produce verification witnesses in the format discussed in Sect. 3, a C-to-Btor2 witness translator, and the witness validator Btor2-Val. Figure 5 shows the translation and validation flows for correctness (in Fig. 5a) and violation witnesses (in Fig. 5b). Both the translator and the validator Btor2-Val for Btor2 witnesses are implemented in Python 3. Btor2-Val is based on a portfolio of hardware verifiers and simulators, with different tools coordinated by the cooperative-verification framework CoVeriTeam [28].

4.1 Validating Correctness Witnesses

Given a safe Btor2 circuit, its translated C program, and a correctness witness produced by some software verifier, Btor2-Cert certifies the results of the software verifier in two steps, as depicted in Fig. 5a. In the first step of witness translation, Btor2-Cert extracts the invariant at the loop head of the C program and represents it as a Btor2 circuit. The Btor2 circuit is named a *witness circuit* and refers to the state variables of the original circuit from its primary inputs. Second, in the validation step, Btor2-Val takes as input the original circuit, the witness circuit, and a user-defined parameter called *invariant quality* that specifies the level of strictness imposed on the invariant. Btor2-Val offers three levels of invariant quality to users, based on which it instruments the original circuit. Hardware verifiers are invoked on the *instrumented circuit* and will deem it safe if the invariant meets the specified invariant quality for reconstructing a safety proof. The details of validating correctness witnesses are presented in Sect. 5.

[2] https://gitlab.com/sosy-lab/software/btor2-cert

4.2 Validating Violation Witnesses

Given an unsafe Btor2 circuit, its translated C program, and a violation witness produced by some software verifier, Btor2-Cert certifies the results of the software verifier in two steps, as depicted in Fig. 5b. In the first step of witness translation, Btor2-Cert extracts the values for external inputs and uninitialized states from the software violation witness and encodes the information as a Btor2 violation witness [7]. Second, in the validation step, Btor2-Val invokes BtorSim [26], a simulator for Btor2 circuits, to decide whether the Btor2 violation witness can trigger a bug in the original circuit. The details of validating violation witnesses are presented in Sect. 6.

5 Certifying Results of Software Verifiers: Correctness

In this section, we describe how Btor2-Cert certifies verification results for **safe** verification tasks. The Btor2 circuit and its translated C program in Fig. 2 as well as the software correctness witness in Fig. 3 will be used to explain the translation and validation of correctness witnesses, as outlined in Fig. 5a.

5.1 Witness Translation

Given a software correctness witness with a predicate annotated at the loop head of the translated C program, which some software verifier claims to be an invariant,[3] Btor2-Cert considers the predicate as a *candidate invariant* for the original Btor2 circuit and extracts it to reconstruct a safety proof. We encode the candidate invariant, written as an expression in the programming language C, into a combinational Btor2 circuit whose inputs refer to the state variables of the original Btor2 circuit and unique output asserts the predicate. Translating C expressions into Btor2 circuits is feasible thanks to the word-level data sorts and operations in the Btor2 language [7]. We name the combinational Btor2 circuit a *witness circuit* and refer to it as a Btor2 correctness witness. Note that our notion of a witness circuit is different from Certifaiger's definition of a *k-witness circuit* [29], which is a sequential circuit simulating *k*-step execution of the original circuit in one step. Figure 6 shows the witness circuit generated from the software correctness witness in Fig. 3. The input defined in line `5` refers to state variable `b` of the Btor2 circuit in Fig. 2a. The output defined in line `9` asserts the candidate invariant `b >= 0 && b <= 1`.

```
1 sort bitvec 8
2 sort bitvec 1
3 zero 1
4 one 1
5 input 1 ; state "b"
6 ugte 2 5 3 ; b >= 0
7 ulte 2 5 4 ; b <= 1
8 and 2 6 7
9 output 8
```

Fig. 6: A witness circuit

5.2 Witness Validation via Verification

Following the idea of validation via verification [27], the validator Btor2-Val in Btor2-Cert checks Btor2 correctness witnesses by instrumenting the original circuit with the witness circuit and invoking hardware model checkers. It distinguishes three levels of quality for a candidate invariant computed by software

[3] Many mature verifiers in SV-COMP derive invariants at loop-head locations.

Table 1: Candidate invariants at the loop head of the program in Fig. 2b

Predicate	Quality	Reason
$\bot$	not invariant	$\mathcal{M} \models Inv$ fails.
$\top$	invariant but unsafe	$Inv \Rightarrow P$ fails.
`b!=2`	safe invariant but not inductive	$Inv(s) \wedge TR(s, s') \Rightarrow Inv(s')$ fails.
`b>=0 && b<=1`	safe and inductive invariant	All checks succeed.

verifiers. According to the notation introduced in Sect. 3, we denote the state-transition system of the original Btor2 circuit by $\mathcal{M}$, with initial states $I(s)$, a transition relation $TR(s, s')$, and a safety property $P(s)$. A predicate $Inv(s)$ is

- an invariant if $\mathcal{M} \models Inv$,
- a *safe* invariant if $\mathcal{M} \models Inv$ and $Inv(s) \Rightarrow P(s)$, and
- a *safe* and *inductive* invariant if (1) $Inv(s) \Rightarrow P(s)$, (2) $I(s) \Rightarrow Inv(s)$, and (3) $Inv(s) \wedge TR(s, s') \Rightarrow Inv(s')$.

In the literature [29], the three conditions for safe and inductive invariants are also named *consistency*, *initiation*, and *consecution*, respectively. Table 1 shows four predicates and highlights their respective quality as an invariant at the loop head of the program in Fig. 2b (P is the negated error condition).

Btor2-Val takes the original Btor2 circuit, the witness circuit, and a user-specified invariant quality for the correctness witness as input and instruments the original circuit accordingly. To check if $Inv(s)$ is an invariant helpful to reestablish a proof of P, Btor2-Val combines the witness circuit and the original circuit by connecting the state variables of the original circuit to the corresponding inputs of the witness circuit. That is, Btor2-Val builds a circuit that encodes $\mathcal{M} \models Inv \wedge P$. The instrumented circuit is given to hardware model checkers, which will utilize the information provided by the witness circuit to find a proof of correctness or refute the predicate if it is not an invariant. Note that the verification time of the instrumented circuit is expected to be shorter than that of the original circuit because the predicate can guide the search of hardware model checkers.

To implement the consistency, initiation, and consecution checks for safe or inductive invariants, Btor2-Val also relies on circuit instrumentation and hardware model checkers. While the three checks are not model checking but satisfiability in essence, it is convenient to encode them as combinational Btor2 circuits. Moreover, some hardware model checkers, such as ABC [50], can simplify the circuits before performing satisfiability solving, which is usually faster than solving the queries directly with satisfiability solvers.

6 Certifying Results of Software Verifiers: Violation

In this section, we describe how Btor2-Cert certifies verification results for **unsafe** verification tasks. The unsafe versions of the Btor2 circuit and its translated C program in Fig. 2 with the state variable `b` being uninitialized (namely, with line `10` in Fig. 2a and line `8` in Fig. 2b commented out) as well as

the software violation witness in Fig. 4 will be used to explain the translation and validation of violation witnesses, as outlined in Fig. 5b.

The Btor2 language defines a format for violation witnesses [7]. A Btor2 violation witness contains a sequence of input values fed to the Btor2 circuit in each cycle and the initial values for uninitialized state variables. Figure 7 shows an example violation witness for the unsafe version of the Btor2 circuit in Fig. 2a. It demonstrates how to trigger the error specified by the 0th `bad` statement (indicted by `b0`) via giving the initial value 2 to the 1st state variable `b` (under `#0`; `a` is the 0th state variable) and 42 to the 0th input `in` in the 2nd cycle (indicated by `@2`). The simulator BtorSim [26] takes a Btor2 circuit and a Btor2 violation witness and executes the circuit with the values for inputs and states in the witness. It confirms the violation witness if an error is triggered. The violation witness in Fig. 7 does not specify input values in the first two cycles because they are irrelevant to the error. In this case, BtorSim will assume the unspecified values to be zero.

6.1 Witness Translation

Given a software violation witness of the translated C program, Btor2-Cert extracts the conditions over program variables from the protocol automaton. These conditions are used by the software violation witness to prune out irrelevant program paths and highlight an error path. Btor2-Cert uses such information to give values to the corresponding Btor2 inputs and state variables in the form of a Btor2 violation witness. For example, the software violation witness in Fig. 4 will be translated to the Btor2 violation witness in Fig. 7.

```
sat
b0
#0
1 00000010 ; b==2
@0
@1
@2
0 00101010 ; in==42
.
```

Fig. 7: A Btor2 violation witness

6.2 Witness Validation via Execution

Following the idea of execution-based witness validation [47], Btor2-Val checks Btor2 violation witnesses by invoking the simulator BtorSim on the original Btor2 circuit and the translated Btor2 violation witness. An advantage of execution-based witness validation is its speed: In our evaluation, Btor2-Val was able to validate Btor2 violation witnesses translated from software violation witnesses much faster than software verifiers for finding the bugs. The speed of Btor2-Val minimizes the overhead to validate the alarms reported by software verifiers and makes the results of software verifiers more trustworthy and transparent for hardware designers.

7 Evaluation

To address the open questions highlighted in Sect. 1.1, we evaluated the proposed certifying hardware-verification framework Btor2-Cert on more than 1 000 Btor2 circuits and the witness validator Btor2-Val prepended with witness translation against the top contenders in the witness-validation track of SV-COMP 2023 [24]. Our experiment is designed to answer the following research questions:

- **RQ1**: Can Btor2-Cert translate software witnesses to Btor2 witnesses?
- **RQ2**: Is Btor2-Val prepended with witness translation **effective** compared to state-of-the-art software witness validators?
- **RQ3**: Is Btor2-Val prepended with witness translation **efficient** compared to state-of-the-art software witness validators?
- **RQ4**: Is the run-time consumed by witness validators shorter than the run-time consumed by software verifiers?
- **RQ5**: Can Btor2-Cert complement conventional hardware model checking by providing additional certified verification results?

7.1 Benchmark Set

We executed our experiments on a benchmark set consisting of 1214 safety-verification tasks of Btor2 circuits, among which 758 are safe and 456 are unsafe. The verification tasks are collected from HWMCC as well as other sources and were used to compare the performance of hardware and software model checkers [5].

7.2 Experimental Settings

All experiments were conducted on machines running Ubuntu 22.04 (64 bit), each with a 3.4 GHz CPU (Intel Xeon E3-1230 v5) with 8 processing units and 33 GB of RAM. The resource limits imposed on verifying translated C programs and validating generated witnesses are both set to 2 CPU cores, 15 min of CPU time, and 15 GB of RAM. We used BenchExec [53] to ensure reliable resource measurement and reproducible results. Btor2-Cert uses Btor2C at commit `36c1ad52` for translating a Btor2 circuit to a C program. In our experiment, we configure the witness validator Btor2-Val to use the PDR [54] implementation in ABC [50] at commit `65ccd3cc` and BtorSim [26] as the underlying hardware model checker and simulator, respectively.[4] We also tried AVR [51] for validating correctness witnesses, but it encountered errors on many instrumented circuits even though the circuits are syntactically valid according to Btor2Tools [26].

7.3 Evaluated Verifiers and Validators

To verify the translated C programs, we used CPAchecker [12] at revision `44619` and UAutomizer [15] at commit `6fd36663` on safe tasks because they are good at constructing invariants in the competitions. We configured CPAchecker to run four algorithms based on Craig interpolation [55], including IMC [56, 57], ISMC [58], Impact [59], and predicate abstraction [60]. On unsafe tasks, we evaluated the BMC [61] implementations in CPAchecker, Cbmc [13], and Esbmc [14] because BMC is the prevailing technique for bug hunting. Both Cbmc and Esbmc were downloaded from the archiving repository of SV-COMP 2023 [52]. For UAutomizer, we used its default settings in SV-COMP for both safe and unsafe tasks.

To evaluate Btor2-Val, we prepended it with the witness-translation step and compared the combination, which takes software witnesses as input, to validators for software witnesses. For correctness witnesses, we evaluated the first place

[4] As ABC works on the bit level, we bit-blasted Btor2 circuits into the AIGER format with Btor2AIGER [26] before invoking ABC.

winner UAutomizer of the witness-validation track in SV-COMP 2023 [24]. We also used an emerging validator LIV [46] at commit cf736e45, which decomposes a program into straight-line sub-programs to check inductive invariants. We cannot compare Btor2-Val to Certifaiger [29, 30] because Certifaiger consumes a candidate inductive length as input, while Btor2-Val expects an invariant from the witnesses. For violation witnesses, we compared Btor2-Val to execution-based validators [47] CPA-w2t and FShell-w2t. The former is of the same version as CPAchecker (i.e., at revision 44619) and the latter was downloaded from the tool archive of SV-COMP 2023 [52]. We also evaluated MetaVal [27], a tool using validation via verification, but it did not terminate when instrumenting the translated C programs and failed to validate any witness in our experiment.

7.4 Results

RQ1: SW-to-HW Witness Translation. The upper part of Table 2 (resp. Table 3) shows the numbers of correctness (resp. violation) witnesses produced by the software verifiers and those successfully translated by the witness translator in Btor2-Cert. Table 2 additionally shows in its 2nd row the numbers of software witnesses with candidate invariants annotated to the loop head of a translated C program. About 97 % of the candidate invariants in software correctness witnesses can be translated to Btor2 witness circuits. The CPAchecker's 14 candidate invariants that cannot be translated were due to the C-expression parser[5] exceeding the time limit when constructing abstract syntax trees. This is a technical limitation orthogonal to the proposed approach. Furthermore, all 4 candidate invariants of UAutomizer that could not be translated refer to undeclared program variables, rendering the witnesses to be syntactically incorrect.[6]

For software violation witnesses, all of them were successfully translated by Btor2-Cert. The median translation time was below 2 s for both correctness and violation witnesses. Moreover, measured by the number of lines of a Btor2 witness, the translated correctness witnesses have a median size of 321, and the violation witnesses have a median size of 308. The results show the feasibility to translate and represent the information found by software verifiers in a native hardware-modeling format.

RQ2: Effectiveness of Btor2-Val. The lower part of Table 2 (resp. Table 3) summarizes the numbers of correctness (resp. violation) witnesses that were validated by Btor2-Val and the compared validators.

Btor2-Val was able to validate the correctness witnesses produced by both CPAchecker and UAutomizer. When configured to accept safe and inductive invariants (recall the three levels of invariant quality in Sect. 5), it validates 329 out of 576 correctness witnesses translated to Btor2 witness circuits. In contrast, UAutomizer, the winner of the witness-validation track in SV-COMP 2023 [24], was not able to validate any correctness witness produced by CPAchecker (the corresponding cells are marked as "-"). LIV is designed to confirm safe and inductive invariants [46] and accepted 305 correctness witnesses in total, similar

[5] Btor2-Cert uses pycparser 2.21 (https://github.com/eliben/pycparser).
[6] https://github.com/ultimate-pa/ultimate/issues/660

Table 2: Summary of results on validating correctness witnesses

Val. \ Verif.	CPAchecker IMC	CPAchecker ISMC	CPAchecker Impact	CPAchecker PredAbs	UAutomizer	Sum of each analysis: accepted	rejected	others
(proofs)	119	85	155	182	79	620	-	-
w/ candidate inv.	114	79	148	178	75	594	-	-
translated	113	79	139	174	71	576	-	-
Bt-Val invariant	77	66	117	119	67	446	105	69
Bt-Val safe	27	47	90	118	45	327	228	65
Bt-Val safe & inductive	28	47	90	118	46	329	243	48
LIV	15	32	95	122	41	305	252	63
UAutomizer	-	-	-	-	74	74	2	3

Table 3: Summary of results on validating violation witnesses

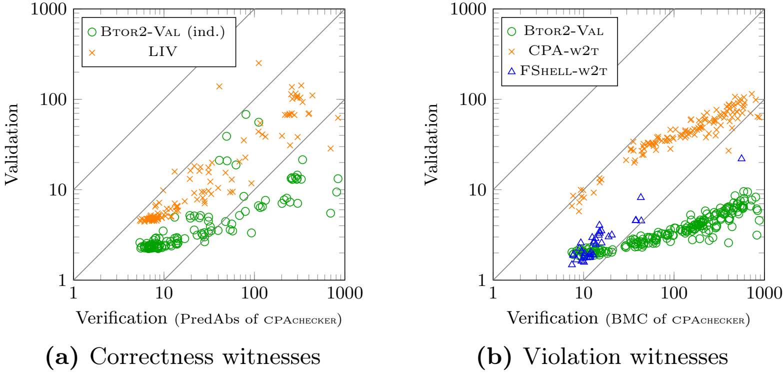
Val. \ Verif.	Cbmc	CPAchecker	Esbmc	UAutomizer	Sum of each analysis: accepted	rejected	others
(alarms)	369	197	302	31	899	-	-
Btor2-Val	59	197	295	27	578	321	0
CPA-w2t	0	122	0	0	122	-	777
FShell-w2t	44	38	44	24	150	-	749

to Btor2-Val. Btor2-Val and LIV agreed on the majority of the correctness witnesses, and the cases where they computed different verdicts were caused by a bug[7] in LIV, which has been fixed by its developers. The results show that Btor2-Val is more robust than UAutomizer and achieves similar effectiveness as LIV. We manually inspected several witnesses rejected by both Btor2-Val and LIV and found that they indeed contain incorrect candidate invariants that do not overapproximate the reachable state spaces. Such invalid invariants might be caused by bugs in the conversion step of software verifiers from its internal formula representation back to the programming language C.

Table 2 also reports the results when Btor2-Val is configured to accept correctness witnesses with different levels of invariant quality. Overall, 77 % of the candidate invariants derived by software verifiers passed the invariant check of Btor2-Val, but only 57 % are deemed safe and inductive. As expected, the number of rejections increases with the strictness for invariant quality. However, there are 2 instances in Table 2 that passed the level "safe & inductive" but were not confirmed at the level "safe" by Btor2-Val. Such cases occurred because ABC, the backend verifier of Btor2-Val, ran into timeout when performing model checking, whereas the consistency, initiation, and consecution checks based on satisfiability easily went through. Among the four interpolation-based algorithms in CPAchecker, predicate abstraction is the best in terms of invariant quality: It generated the most safe and inductive invariants. The results demonstrate the unique value of Btor2-Val to quantify the quality of invariants derived by software verifiers.

For violation witnesses, Btor2-Val was far more effective than CPA-w2t and FShell-w2t in our experiment. Among 899 violation witnesses generated by software verifiers, Btor2-Val was able to validate 578 cases; It rejected

[7] https://gitlab.com/sosy-lab/software/liv/-/issues/2



(a) Correctness witnesses

(b) Violation witnesses

Fig. 8: CPU-time comparison of verification and witness validation (unit: s)

other witnesses because they contain an incomplete or infeasible error path. In comparison, CPA-w2t and FShell-w2t only confirmed 122 and 150 witnesses, respectively. The numbers of rejected witnesses for CPA-w2t and FShell-w2t are not listed in Table 3 as the tools do not distinguish rejection of witnesses from other errors. We also observed that only 11 violation witnesses produced by CPAchecker, Esbmc, and UAutomizer were not validated by Btor2-Val, but witnesses generated by Cbmc suffered from a high rejection rate. This is because the violation witnesses of Cbmc often report an infeasible error path. Moreover, we notice that for many cases, different error paths are printed in Cbmc's violation witnesses and the console logs for its execution.[8] If we extract Btor2 violation witnesses from the console logs instead, Btor2-Val could validate 359 out of the 369 cases where Cbmc found an alarm. The effectiveness of Btor2-Val in confirming translated Btor2 violation witnesses showcases the value of Btor2-Cert because hardware designers can now trust software verifiers to detect bugs in their circuits and obtain a certified test case to trigger an error if software verifiers reported one.

RQ3: Efficiency of Btor2-Val. We compared the CPU time required for Btor2-Val and other state-of-the-art validators. From our experimental results, Btor2-Val (configured to accept safe and inductive invariants) achieved a median speedup of 2.2× over LIV for correctness witness validation, and a median speedup of 11× and 1.1× over CPA-w2t and FShell-w2t for violation witness validation, respectively. In addition, Fig. 8 shows the scatter plots for the CPU time consumption of the compared validators. A data point (x, y) in the plots corresponds to a case where CPAchecker took x seconds to produce a witness and a validator took y seconds to validate the witness. Observe that most data points of Btor2-Val are below those of other validators. The efficiency of the proposed certifying framework in translating and validating violation witnesses minimizes the overhead to apply software analyzers to find hardware bugs and makes the results of software verifiers trustworthy for hardware designers.

[8] https://github.com/diffblue/cprover-sv-comp/issues/70

RQ4: Verfication versus Validation Time. Figure 8a (resp. Figure 8b) compares the CPU time for CPAchecker to compute a verdict and generate a correctness (resp. violation) witness to the CPU time for a validator to check the witness. We can see that almost all data points are below the diagonal, indicating that validation time is typically shorter than verification time. Such speedup shows that the validators are able to utilize the information in witnesses to reconstruct proofs of correctness or violation more efficiently than verifying the task from scratch.

RQ5: Complementing HW Model Checking with Btor2-Cert. The empirical evaluation in the TACAS 2023 publication [5] on Btor2C demonstrates that software verifiers are able to complement the state-of-the-art hardware model checkers by finding more bugs and uniquely solving dozens of tasks. We take a step further and investigate whether the verification results of those additional alarms and uniquely solved tasks can be certified by Btor2-Cert.

Btor2-Cert **certified** 37, 1, and 4 alarms found by the BMC implementations of Cbmc, CPAchecker, and Esbmc, respectively, which cannot be detected by the BMC implementation of ABC.[9] The additional alarms found by Cbmc alone account up to 8 % of unsafe tasks in our benchmark set. With the help of Btor2-Cert, the violation witnesses generated by software verifiers can be translated to Btor2 witnesses and validated by BtorSim. That is, the property violation reported by software verifiers can be replayed fully in the hardware domain, demonstrating the unique ability of Btor2-Cert to provide trustworthy verification results obtained by software analyzers.

For property satisfaction, although the previous study shows that software verifiers are not as good at finding proofs for correctness as their hardware counterparts, we still observed a case where ABC (the backend verifier used by Btor2-Val) went into timeout but only required less than 3 s to reconstruct a proof using the invariant generated by CPAchecker, and another case with a 5× run-time speedup.

Summaries of Results. From the reported results, we conclude that (1) software witnesses can be translated to hardware witnesses (Table 2 and Table 3), (2) Btor2-Val is effective (Table 2 and Table 3) and efficient (Fig. 8), (3) witness validation by Btor2-Val consumes less time than software verification (Fig. 8), and (4) Btor2-Cert complements state-of-the-art hardware model checkers.

As a by-product of this work, our intensive investigation of software witnesses led to the discovery of several bugs in software verifiers. We reported the issues to the developers of the tools, and some of the bugs have been fixed. A complete list of issues that we found in software analyzers during this project is available on the supplementary webpage [62].

7.5 Threats to Validity

For **external validity**, our claims are established on a large set of Btor2 circuits to increase confidence, but it is unclear if they will hold on tasks with different

[9] We considered the 359 validated witnesses translated from console logs of Cbmc.

features that are not covered in the used benchmark set. For **construct validity**, we report that witness validation is faster than verification, but validation and verification were done on behaviorally equivalent but syntactically different models (namely, a Btor2 circuit vs. a C program). While the setting is not exactly the same as in a previous publication [4], it is necessary because our experiment is designed to investigate how information in software witnesses can be used by hardware analyzers. We compared Btor2-Val prepended with witness translation to software witness validators. The former also uses the original Btor2 circuit as input, but the validators for software do not leverage circuit information. We performed the comparison this way because the hardware witness validator Certifaiger [29] does not accept an invariant as input. For **internal validity**, we ran the experiments with the popular benchmarking framework BenchExec [53] to guarantee reproducibility.

8 Conclusion

Validating verification results is vital to make formal methods applicable in practice, as it reinforces the trust of users and offers more insights into the analyzed model. In this manuscript, we proposed Btor2-Cert, a certifying and validating hardware-verification framework built upon translators and software analyzers. Btor2-Cert is an open-source toolchain, involving the Btor2-to-C translator Btor2C, certifying verifiers for C programs, a C-to-Btor2 witness translator, the Btor2 simulator BtorSim, and the validator Btor2-Val. We evaluated Btor2-Cert's capability of transferring the information across software and hardware analyzers and providing certified verification results on a large benchmark set. By employing software model checkers for hardware verification, we identified and certified 8 % of the unsafe tasks in our benchmark set that the state-of-the-art conventional hardware model checker ABC overlooked. For future work, we will augment Btor2-Cert to accommodate temporal decomposition [48], a preprocessing technique used to simplify sequential circuits before model checking. Such extension [31] has been made to k-inductiveness validators [29, 30].

Data-Availability Statement. All verification tasks, tools, and experimental results from our evaluation are available in the reproduction artifact [63]. A previous version [64] of the reproduction package was reviewed by the Artifact Evaluation Committee. The updated version [63] fixes some bugs in the witness translator. More information is available on the supplementary webpage [62].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY). Zsófia Ádám is supported partially by the Doctoral Excellence Fellowship Programme, which is funded by the National Research, Development and Innovation Fund of the Ministry of Culture and Innovation, and the Budapest University of Technology and Economics, under a grant agreement with the National Research, Development and Innovation Office.

References

1. 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
2. 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
3. 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
4. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
5. 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. pp. 1–21. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_12
6. Tafese, J., Garcia-Contreras, I., Gurfinkel, A.: Btor2MLIR: A format and toolchain for hardware verification pp. 55–63 (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_13
7. 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
8. 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
9. Biere, A., Froleyks, N., Preiner, M.: 11th Hardware Model Checking Competition (HWMCC 2020). http://fmv.jku.at/hwmcc20/, accessed: 2023-01-29
10. 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
11. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO. pp. 75–88. IEEE (2004). https://doi.org/10.1109/CGO.2004.1281665
12. 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
13. 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
14. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: Proc. ASE. pp. 888–891. ACM (2018). https://doi.org/10.1145/3238147.3240481
15. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
16. 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). https://dl.acm.org/doi/10.5555/1855741.1855756

17. 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
18. 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
19. 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
20. 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
21. 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
22. IEEE standard for Verilog hardware description language (2006). https://doi.org/10.1109/IEEESTD.2006.99495
23. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An open-source tool for symbolic model checking. In: Proc. CAV. pp. 359–364. LNCS 2404, Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
24. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
25. 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
26. 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
27. 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
28. 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
29. Yu, E., Biere, A., Heljanko, K.: Progress in certifying hardware model checking results. In: Proc. CAV. pp. 363–386. LNCS 12760, Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_17
30. Yu, E., Froleyks, N., Biere, A., Heljanko, K.: Stratified certification for k-induction. In: Proc. FMCAD. pp. 59–64. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_11
31. Yu, E., Froleyks, N., Biere, A., Heljanko, K.: Towards compositional hardware model checking certification. In: Proc. FMCAD. pp. 1–11. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_12
32. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. FMCAD, pp. 127–144. LNCS 1954, Springer (2000). https://doi.org/10.1007/3-540-40922-X_8
33. Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M.: SAT competition 2020. Artif. Intell. **301**, 103572:1–103572:25 (2021). https://doi.org/10.1016/j.artint.2021.103572

34. Järvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine **33**(1) (2012). https://doi.org/10.1609/aimag.v33i1.2395
35. Heule, M.J.H.: The DRAT format and DRAT-TRIM checker. CoRR **1610**(06229) (October 2016). https://doi.org/10.48550/arXiv.1610.06229
36. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2020). https://doi.org/10.1007/s10817-019-09525-z
37. Wetzler, N., Heule, M.J.H., Jr., W.A.H.: DRAT-TRIM: Efficient checking and trimming using expressive clausal proofs. In: Proc. SAT. pp. 422–429. LNCS 8561, Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31
38. Bury, G., Bobot, F.: Verifying models with DOLMEN. In: Proc. SMT Workshop. CEUR Workshop Proceedings, CEUR (2023). https://ceur-ws.org/Vol-3429/short9.pdf
39. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF - (tool presentation). In: Proc. SAT. pp. 430–435. LNCS 7317, Springer (2012). https://doi.org/10.1007/978-3-642-31612-8_33
40. Balabanov, V., Jiang, J.H.R.: Unified QBF certification and its applications. Formal Methods Syst. Des. **41**(1), 45–65 (2012). https://doi.org/10.1007/s10703-012-0152-6
41. Namjoshi, K.S.: Certifying model checkers. In: Proc. CAV. pp. 2–13. LNCS 2102, Springer (2001). https://doi.org/10.1007/3-540-44585-4_2
42. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: Proc. CADE. pp. 105–108. LNCS 9195, Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_6
43. Sternagel, C., Thiemann, R.: The certification problem format. In: Proc. UITP. pp. 61–72. EPTCS 167, EPTCS (2014). https://doi.org/10.4204/EPTCS.167.8
44. Griggio, A., Roveri, M., Tonetta, S.: Certifying proofs for LTL model checking. In: Proc. FMCAD. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603022
45. Kahsai, T., Tinelli, C.: PKIND: A parallel k-induction based model checker. In: Proc. Int. Workshop on Parallel and Distributed Methods in Verification. pp. 55–62. EPTCS 72, EPTCS (2011). https://doi.org/10.4204/EPTCS.72.6
46. 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
47. 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
48. Case, M.L., Mony, H., Baumgartner, J., Kanzelman, R.: Enhanced verification by temporal decomposition. In: Proc. FMCAD. pp. 17–24. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351146
49. 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
50. 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
51. 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

52. Beyer, D.: Verifiers and validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7627829
53. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
54. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proc. FMCAD. pp. 125–134. FMCAD Inc. (2011). https://dl.acm.org/doi/10.5555/2157654.2157675
55. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593
56. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
57. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. arXiv/CoRR **2208**(05046) (July 2022). https://doi.org/10.48550/arXiv.2208.05046
58. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Proc. FMCAD. pp. 1–8. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351148
59. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_14
60. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
61. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
62. Ádám, Z., Beyer, D., Chien, P.C., Lee, N.Z., Sirrenberg, N.: Supplementary webpage for TACAS 2024 article 'Btor2-Cert: A certifying hardware-verification framework using software analyzers'. https://www.sosy-lab.org/research/btor2-cert/
63. Ádám, Z., Beyer, D., Chien, P.C., Lee, N.Z., Sirrenberg, N.: Reproduction package for TACAS 2024 article 'Btor2-Cert: A certifying hardware-verification framework using software analyzers'. Zenodo (2023). https://doi.org/10.5281/zenodo.10548597
64. Ádám, Z., Beyer, D., Chien, P.C., Lee, N.Z., Sirrenberg, N.: Reproduction package for TACAS 2024 submission 'Btor2-Cert: A certifying hardware-verification framework using software analyzers'. Zenodo (2023). https://doi.org/10.5281/zenodo.10013059

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.