Noname manuscript No.
(will be inserted by the editor)

Bridging Hardware and Software Analysis with Btor2C: A Word-Level-Circuit-to-C Translator

(Extended Version)

Salih Ates[1] · **Dirk Beyer**[1] · **Po-Chun Chien**[1] · **Nian-Ze Lee**[1,2]

the date of receipt and acceptance should be inserted later

Abstract Across the broad research field concerned with analyzing computing systems, algorithms and tools revolve around the modeling languages used to describe the systems, hindering their applications to similar problems of systems in other modeling languages. For example, the research communities for formal verification and testing of hardware and software share common theoretical foundations and solving methods, including symbolic encoding, satisfiability solving, and abstraction refinement. Nevertheless, it requires significant effort for one community to benefit from the advancements of the other, as analyzers assume different modeling languages for input instances. To bridge the gap between hardware and software analysis, we propose Btor2C, a translator from word-level sequential circuits in the Btor2 language to C programs. We choose the Btor2 language as frontend because its simple syntax and bit-precise semantics make it a suitable *intermediate representation* for analysis purposes. Using Btor2C, we translate Btor2 circuits from the Hardware Model Checking Competitions into C programs and analyze them using tools from the Intl. Competitions on Software Verification and Testing. Our results show that software analyzers can complement hardware model checkers for enhanced quality assurance: Prominently, the software verifier Cbmc (with Btor2C for preprocessing) found more bugs than the best hardware model checkers ABC and AVR in our experiment.

Keywords Hardware-model translation · C · Word-level circuit · Intermediate representation · Formal verification · Testing · Btor2 · SMT · SAT · SV-COMP

A conference version is published at TACAS 2023 [1].

[1]LMU Munich, Munich, Germany
[2]National Taiwan University, Taipei, Taiwan

1 Introduction

Computing systems support our society and economy, so meticulous analysis is needed to ensure their correct functionality and security. A vast amount of effort has been invested to develop analysis techniques for different modeling languages used to describe computing systems. Because of the ever-increasing system complexity and applications in safety-critical missions, it is crucial to join forces of approaches for different types of systems, including VLSI circuits, software programs, and cyber-physical components, to guarantee the quality and correctness of computing systems.

Formal verification and testing are two active research areas for quality assurance of computing systems. The former decides with mathematical rigor whether a computational model conforms to a specification. The latter aims to generate input patterns and execute a model on a test suite to observe irregular output responses. Studies for formal verification or testing usually focus on specific computational models, e.g., sequential circuits (hardware) or programs (software). Tool competitions are also established based on the modeling languages for input instances, such as the Btor2 [2] language used in the Hardware Model Checking Competitions (HWMCC) [3, 4], or the programming language C assumed by the Competitions on Software Verification (SV-COMP) [5] and Testing (Test-Comp) [6]. Unfortunately, such distinction hinders mutual learning and cooperation between the two communities.

1.1 Our Motivation and Contributions

For the hardware community to conveniently benefit from state-of-the-art software-analysis techniques, we

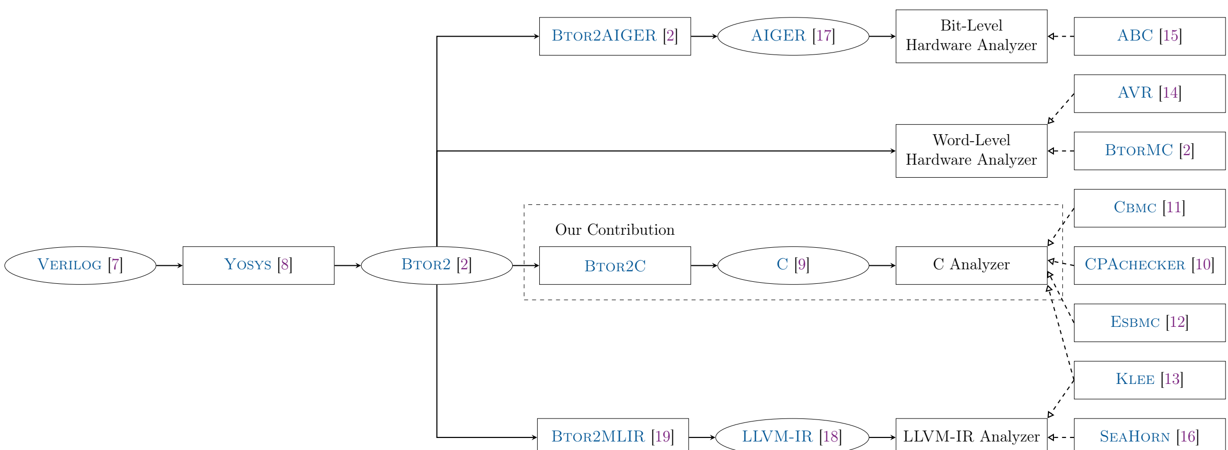


Fig. 1: Enhancing hardware analysis via standalone translators and tools for software analysis

aim at developing a lightweight yet effective translation flow to bridge the gap between hardware and software analysis. There exist several translators from hardware to software in the literature [20, 21, 22, 23], mostly using the Verilog language [7] as frontend. Verilog is a general-purpose hardware description language, and thus, a comprehensive frontend for Verilog requires tremendous engineering effort. Moreover, Verilog has rather complicated syntax and semantics, which might increase the burden on the translation flow.

To address the complexity of the frontend design, we resort to the Btor2 [2] language, proposed recently to model word-level sequential circuits. A suite Btor2Tools [24] of utility tools is also provided for conveniently parsing, simulating, and bit-blasting Btor2 circuits (to the bit-level format AIGER [17]). We emphasize the following two benefits of using Btor2 as the source language of translation over Verilog. First, Btor2 provides simple yet sufficient operations over bit-vectors and arrays. The simplicity makes it an appropriate *intermediate representation* for formal verification and testing, as the operations are suitable for satisfiability solvers. Second, Btor2 is the input format used in the HWMCC, with extensive tool support and a large collection of benchmarking tasks for empirical evaluation. In practice, a Verilog circuit can be translated to Btor2 via Yosys [8], an open-source Verilog synthesis tool. Therefore, using Btor2 as frontend does not restrict the applicability of the translation flow.

Now that the source language is decided, our next consideration is: Should we integrate the translation flow in software analyzers or implement a standalone translator that does the job for all tools? While a customized Btor2 parser may allow performance fine-tuning, reinventing the wheel in every software analyzer is time-consuming and error-prone. Instead, a standalone translator is more modular and robust because it separates the concerns of translation and analysis. Taking the approach of standalone translators, we use the programming language C [9] as the target language and build a Btor2-to-C translator Btor2C. Using Btor2C for preprocessing, any software analyzer for C programs (e.g., from the 76 tools in the literature [25]) can in principle be applied to Btor2 circuits. As opposed to using Verilog as frontend, the simplicity of the Btor2 language helps to generate C programs suitable for the backend analysis, as will be shown in Sect. 6 via comparison with the Verilog-to-C translator v2c [21].

The contributions of the paper are as follows:

The First Btor2-to-Software Translator. To bridge the gap between hardware and software analysis, we design and implement Btor2C, the first translator from hardware to software taking the word-level hardware modeling language Btor2 [2] as input. Specifically, Btor2C accepts a Btor2 circuit and produces a behaviorally equivalent C program.

As depicted in Fig. 1, Btor2C (with the help of Yosys) makes off-the-shelf software verifiers, such as Cbmc [11], CPAchecker [26], Esbmc [12], and testers, such as Klee [13], readily available for Verilog circuits. Therefore, hardware developers will have more choices beyond conventional bit-level (e.g., ABC [15]) and word-level model checkers (e.g., AVR [14] and BtorMC [2]). Shortly after the first publication of Btor2C [1], another Btor2-to-software translator Btor2MLIR [19] was introduced, translating Btor2 circuits to LLVM-IR programs, which can be consumed by Llvm model checkers like SeaHorn [16].

An Extensive Evaluation of Hardware and Software Analyzers. While tools for hardware and software verification share common theoretical foundations,

it is not straightforward to compare them in a head-to-head manner. Smaller experiments on VERILOG circuits via the translator V2C [21] have been reported for bounded [27] and unbounded [28] formal verification, but a large-scale evaluation and detailed analysis of results remained scarce in the literature. By building a translator for the BTOR2 language, more than a thousand benchmarking tasks from the HWMCC are at our immediate disposal. To draw a more reliable conclusion on performance comparison, we conduct experiments involving state-of-the-art bit-level and word-level hardware model checkers, software verifiers from SV-COMP, and software testers from Test-Comp on the HWMCC benchmark set. Our results show that software-analysis techniques can complement hardware model checkers by finding bugs and safety proofs that the state-of-the-art hardware model checkers cannot deliver.

New Verification Tasks for Benchmarking Software Analyzers. Applying BTOR2C to the HWMCC benchmark set, we submitted 1 224 new tasks[1] to sv-benchmarks, the benchmark collection used by many researchers, including the tool competitions SV-COMP and Test-Comp. Developers of software analyzers can now assess their tools using the hardware-analysis counterparts as a new baseline.

The above contributions are *novel* because of the first BTOR2-to-software translator, the new way to build hardware analyzers with BTOR2C and software analyzers, the new insights from our experiments, and the new benchmarking tasks for software analyzers. The contributions are *significant* because they reveal that hardware analysis can be improved and complemented by applying off-the-shelf software analyzers. BTOR2C makes software analysis more accessible to the entire research community, not limited to the hardware domain, as BTOR2 can also be used as an intermediate representation for other models (e.g., MoXI [29]).

1.2 Example

Figure 2 illustrates the proposed translator BTOR2C on an example. A circuit whose state is a bit-vector of width 3 is given in BTOR2 format in Fig. 2a. The bit-vector is initialized to 0 (lines 2 to 4). In every iteration, the value of the bit-vector will be incremented by the value of the external input (lines 5 to 6) and then decremented by 1 (lines 7 to 8). The circuit reaches a bad state (i.e., violates the safety property) if the value of the bit-vector equals `0b111` (lines 12 to 13).

Figure 2b shows the translated C program. BTOR2C (variable-naming convention used by BTOR2C explained

```
1 sort bitvec 3
2 zero 1
3 state 1
4 init 1 3 2
5 input 1
6 add 1 3 5
7 one 1
8 sub 1 6 7
9 next 1 3 8
10 ones 1
11 sort bitvec 1
12 eq 11 3 10
13 bad 12
```

(a) BTOR2 circuit

```
1  extern void abort(void);
2  extern unsigned char nondet_uchar();
3  void main() {
4    typedef unsigned char SORT_1;
5    typedef unsigned char SORT_11;
6    const SORT_1 var_2 = 0b000;
7    const SORT_1 var_7 = 0b001;
8    const SORT_1 var_10 = 0b111;
9    SORT_1 state_3 = var_2;
10   for (;;) {
11     SORT_1 input_5 = nondet_uchar();
12     input_5 = input_5 & 0b111;
13     SORT_1 var_6 = state_3 + input_5;
14     var_6 = var_6 & 0b111;
15     SORT_1 var_8 = var_6 - var_7;
16     var_8 = var_8 & 0b111;
17     SORT_11 var_12 = state_3 == var_10;
18     SORT_11 bad_13 = var_12;
19     if (bad_13) {
20       ERROR: abort();
21     }
22     state_3 = var_8;
23   }
24 }
```

(b) C program (simplified for illustration)

Fig. 2: An example BTOR2 circuit (a) and its translated C program (b)

in Sect. 4.2) first looks for the sorts used in the BTOR2 circuit. In Fig. 2a, bit-vectors of three bits and one bit are used, and BTOR2C encodes them with the shortest possible unsigned integer type `unsigned char` (lines 4 to 5). After sort declarations, BTOR2C defines constants and initializes circuit states (lines 6 to 9). An infinite loop is created to simulate the circuit's behavior. At the beginning of the loop, the circuit inputs are initialized to nondeterministic values and masked to the range of their respective types (lines 11 to 12; masking with `0b111` since the input has three bits). BTOR2C then evaluates the signals of the circuit in a topological order from the inputs and current states to the outputs and next states (lines 13 to 18). If the safety property is violated (namely, variable `bad_13` evaluates to *true*), the program reaches the error location at line 20. Otherwise, the next-state

[1] Some tasks used in this paper were excluded due to licenses.

value (stored in variable `var_8`) is assigned to the current state (line 22), and another loop iteration follows. Software verifiers can be applied to the translated program in Fig. 2b to check whether the circuit in Fig. 2a conforms to the specified safety property.

2 Related Work

In the literature of hardware and software analysis, researchers have invented tools to translate a model in a source language to a behaviorally equivalent one in a target language, which can be a high-level modeling language used by human designers or an intermediate representation consumed by other tools. Classifying translators by their target languages, we discuss several representative translators as follows.

2.1 Translating to High-Level Modeling Languages

Translating models of digital circuits to imperative programs has been used to speed up hardware simulation [20]. For example, VERILATOR [22] translates a VERILOG circuit to a C program whose compiled executable can be run for fast circuit simulation. In the area of formal verification, VHD2CA [23] translates VHDL [30] circuits to counter automata, which can be represented as C programs and analyzed by software verifiers. V2C [21], a Verilog-to-C translator, translates VERILOG circuits to C programs for bounded [27] and unbounded [28] model checking. Unlike V2C, our hardware-to-software translator BTOR2C uses as frontend the BTOR2 language, which is tailored for formal verification. In Sect. 6, we will compare the performance of software analyzers on two sets of C programs, translated by V2C and BTOR2C, respectively.

The other translation direction from software to hardware has also been studied. For example, C2V [31] translates a single-threaded, non-recursive C program to a VERILOG circuit, which can be verified by hardware model checkers.

2.2 Translating to Intermediate Representations

Intermediate representations (IRs) are languages used by tools to accommodate problem instances described in more abstract languages. Compared to high-level modeling languages used by human designers, IRs usually have restricted syntax but precise semantics to facilitate automatic analysis. For example, AIGER [17] is a bit-level IR for hardware synthesis and verification, and BTOR2 [2] is an IR for word-level model checking. In the compiler infrastructure, LLVM-IR [18] has served as the target language for various high-level programming languages, and MLIR [32] offers a framework to define custom IRs as dialects of LLVM-IR. To leverage the inherent simplicity of IRs, we chose BTOR2 as the frontend of the proposed hardware-to-software translator.

Among numerous translators from hardware circuits to IRs, VERILOG2SMV [33] and VER2SMV [34] translate a VERILOG circuit to the SMV format [35, 36], which can be consumed by model checkers like NUXMV [37]. YOSYS [8], an open-source synthesis and verification framework for VERILOG, can translate a VERILOG circuit to the AIGER or BTOR2 formats. BTOR2MLIR [19], a recent translator that also considers BTOR2 as its source language, proposes an LLVM-IR dialect for BTOR2 in the MLIR framework [32]. A BTOR2 circuit is translated to LLVM-IR via the dialect, and software analyzers consuming LLVM-IR as input, such as SEAHORN [16], SMACK [38], and KLEE [13], can be invoked on the translated LLVM-IR program to analyze the BTOR2 circuit. In Sect. 6, we will compare the combination of BTOR2C and software verifiers for C programs against that of BTOR2MLIR and software verifiers for LLVM-IR programs on verifying BTOR2 circuits.

The other direction, translating software programs to IRs, has also been intensively studied in the literature [11, 18, 39]. Recently, SMV and BTOR2 have been used as IRs to accommodate C programs, which gives rise to software verifiers using techniques or tools from hardware model checking as the underlying engines, such as in KRATOS2 [40] and CPV [41]. In light of the large number of IRs tailored for specific use cases, there have been attempts to design "general-purpose" IRs. For example, VMT [42] and MOXI [29] for transition systems and SV-LIB [43] for imperative programs, extend the SMT-LIB 2 format [44] to capture the model-checking problem of infinite-state systems and provide a common ground to evaluate and compare research advancements. CIRCT [45] uses LLVM-IR and its dialects as an alternative framework for circuit synthesis.

Another example of a translator between modeling formats is HYST [46], which converts hybrid automaton models between the input languages of several verification tools, including SPACEEX [47], FLOW* [48], and DREACH [49]. Although primarily designed for continuous and hybrid systems, HYST exemplifies how translators can facilitate interoperability and comparative analysis across different verification backends.

3 Background

3.1 The Btor2 Language

Btor2 is a bit-precise modeling language for word-level sequential circuits. It can be seen as a generalization of the bit-level AIGER format [17]. The essential ingredients of Btor2 relevant to our discussion in Sect. 4 will be introduced below. For the complete syntax, please refer to its CAV 2018 publication [2].

Each line in a Btor2 file starts with a unique number, used by other lines to identify the entity defined in this line. Such an entity can be either a *sort* or a *node*. A sort is either a bit-vector type of an arbitrary width w, denoted by $\mathcal{B}^w$, or an array type. An array type whose indices and elements are of types $\mathcal{I}$ and $\mathcal{E}$, respectively, is denoted by $\mathcal{A}^{\mathcal{I} \to \mathcal{E}}$. A node can be an *input*, a *state*, or a *result* of an operator over other inputs, states, or results. Inputs are external stimuli given to the Btor2 circuit. Memory elements of the circuit are modeled by states. Usually, indices and elements of arrays types, as well as inputs, have bit-vector types, and states can be of either bit-vector or array types.

Operators are the building blocks of a Btor2 circuit. They take arguments of the prescribed types and guarantee a specific type for the result. The signature of a Btor2 operator is as follows:

```
<node id> <op> <sort id0> <node id1>
                  [<node id2 [node id3]>].
```

It defines a node to be the computation result of the operator `op` on node `id1` and optionally `id2` and `id3`. The result will have sort `id0` and can be accessed by `id`. The operators in Btor2 will be introduced later in Sect. 4 alongside the translation process of Btor2C.

Btor2 also provides constructs like `init`, `next`, and `bad` to describe the safety-reachability problem for sequential circuits. Initial and bad states can be defined by `init` and `bad`, respectively. The transition from one state to another is captured by `next`. In the following, we briefly recap sequential circuits and their model-checking formulation.

3.2 Sequential Circuits and Hardware Model Checking

A sequential circuit is a computational model widely used in the design and analysis of hardware. It consists of a combinational circuit and memory elements. The memory elements store the circuit's state, and the combinational circuit computes the transition of states. The combinational circuit is a directed acyclic graph with vertices being logic gates and edges being wires connecting the gates. If the output pin of gate u is connected to an input pin of gate v, we say that u is a *fan-in* of v, and v is a *fan-out* of u.

The computation of sequential circuits is segmented into consecutive time frames. Before the first time frame starts, the memory elements are typically reset (described by keyword `init` in Btor2). At the beginning of each time frame, the combinational circuit reads the values stored in the memory elements and receives stimuli from the environment. The former is called the *current state* of the circuit, and the latter is called the *external input* in this time frame. Propagating the current state and external input through its logic gates, the combinational circuit computes the output response and the values to be stored in the memory elements (namely, the *next-state* values, described by keyword `next` in Btor2). At the end of the time frame, the next-state values are saved into the memory elements, which become the current state in the next time frame.

The model-checking problem of *reachability safety* for hardware is formulated as follows: Given a sequential circuit and a safety property (usually encoded as an output of the combinational circuit, described by keyword `bad` in Btor2), decide whether the safety property holds on all executions of the sequential circuit. If the property does not hold on some execution, a hardware model checker generates an input sequence to trigger the output, and the sequential circuit is deemed unsafe with respect to the property. Otherwise, the sequential circuit is considered safe, and a model checker might additionally generate (an overapproximation of) the set of reachable states as a correctness witness [50].

3.3 Software Model Checking

The reachability-safety problem for software is formulated similarly as hardware model checking: Given a program and a safety property (usually labeled as an error location in the program), determine whether there is an executable program path that reaches the error location. Although software model checking is in general undecidable, many research efforts have been invested into automated solutions to this problem [51, 52, 53], including predicate abstraction [54, 55, 56, 57], counterexample-guided abstraction refinement (CEGAR) [58, 59], and interpolation [60, 61]. The verification of industry-scale software [62, 63, 64, 65, 66, 67] has been made feasible together by these approaches and the advances in SMT solving [68]. We want to explore how these concepts work on hardware analysis.

```
void main() {
  // 1. Define sorts and constants
  // 2. Initialize states
  for (;;) {
    // 3. Receive external inputs
    // 4. Evaluate intermediate nodes
    // 5. Assume constraints
    // 6. Check safety property
    if (property_violated) {
      ERROR: abort();
    }
    // 7. Assign next-state values
  }
}
```

Fig. 3: A generic program to imitate sequential circuits for reachability safety

4 Translating Btor2 to C

This section describes the translator Btor2C,[2] implemented in the programming language C++ with approximately two thousand lines of code. We first describe the general idea of using C programs to simulate sequential circuits, whose behavior is intrinsically concurrent. The implementations of various Btor2 operators and optimizations in Btor2C are discussed later.

4.1 Simulating Sequential Circuits with C Programs

Sequential circuits work in a concurrent manner: The external input and current state propagate in parallel through the combinational circuitry to produce circuit outputs and next-state values. In contrast, the programming language C is imperative, and hence C programs are generally executed line by line.

To capture the behavior of sequential circuits in the context of reachability safety, Btor2C generates C programs with the generic single-loop program in Fig. 3 as a template. In the generic program, the sorts and constants used in the sequential circuit are defined at the beginning of the `main()` function. Second, the program initializes the circuit's state variables. An endless loop is then used to mimic the state-transition behavior of the circuit throughout time frames: When a loop iteration begins, the circuit receives external inputs and propagate them together with the current-state values to evaluate its intermediate nodes. Note that the evaluation of intermediate nodes must follow a topological order from inputs to outputs to guarantee that the value of a node is evaluated after all its fan-ins are evaluated. After the intermediate nodes are evaluated, constraints (keyword `constraint` in Btor2) of the circuit are added, and the safety property is checked assuming that the constraints hold. If the property is violated, the program exits with an error. Otherwise, the next-state values are stored into the state variables. This generic program reflects the reachability safety for sequential circuits.

The commented blocks in the generic program have to be replaced by C instructions to encode the concurrent computation of the sequential circuit. In the translated C program, Btor2C assigns every node in the input Btor2 circuit a unique variable. Nodes used for state initialization, state transition, or safety properties, are specified by keywords `init`, `next`, or `bad`, respectively. For such a node, a backward depth-first traversal is applied to collect its transitive fan-in cone to exclude irrelevant signals from model checking. Multiple `bad` keywords in a Btor2 circuit are translated to multiple error labels in the C program.

4.2 Variable Naming

We use the unique identification numbers for lines in a Btor2 file to name their corresponding types and variables in the translated C program. Suppose the unique ID of a line is `n`. If the line defines a sort, the corresponding type is named `SORT_n` in the C file. If the line defines a state or an input, the corresponding variable is named `state_n` or `input_n`, respectively. If the line defines a node used for state transition or property evaluation, the corresponding variable is named `next_n` or `bad_n`, respectively, to honor the keywords `next` and `bad`. Nodes defining a Btor2 constant or operation are mapped to variables named `var_n` in the C program.

4.3 Expressing Btor2 Sorts in C

Btor2 supports two sorts: bit-vectors and arrays. A bit-vector type $\mathcal{B}^w$ is represented by the shortest unsigned-integer type whose number of bits is greater than or equal to w. For example, a $\mathcal{B}^3$ type with sort ID `n` is encoded by

```
typedef SORT_n unsigned char;
```

and a $\mathcal{B}^{20}$ type with sort ID `m` is encoded by

```
typedef SORT_m unsigned int;.
```

A Btor2 bit-vector type can have an arbitrary width. If a Btor2 circuit uses a bit-vector type longer than 64 bits, Btor2C cannot translate it to a C program,

[2] https://gitlab.com/sosy-lab/software/btor2c

because no standard C type can accommodate the bit-vector.[3] The missing capability to handle bit-vectors longer than 64 bits is a restriction of Btor2C, but the sacrifice is worthy: By encoding bit-vectors with integer variables, native C operators can be directly applied to implement Btor2 operators, which greatly simplify the analysis of translated programs. As can be seen in Sect. 6, the state-of-the-art software verifiers and testers have a decent performance on the translated programs. In practice, less than 20 % of the collected Btor2 benchmarking circuits have bit-vectors longer than 64 bits, so we consider the restriction acceptable.

For array sorts whose index and element sorts are of bit-vectors in Btor2, Btor2C represents them using static arrays. Suppose the sort ID for an array type $\mathcal{A}^{\mathcal{I}\to\mathcal{E}}$ has sort ID `n`, with index type $\mathcal{I}$ of $\mathcal{B}^w$ and element type $\mathcal{E}$ named `SORT_m`. Then $\mathcal{A}^{\mathcal{I}\to\mathcal{E}}$ is encoded in C as:

```
typedef SORT_m SORT_n[1 << w];,
```

which defines `SORT_n` as an array with 2^w elements of type `SORT_m`.

4.4 Implementing Btor2 Operators in C

The language Btor2 provides various operations, most of which can be easily implemented by the corresponding C operators. Recall that we extend to the next unsigned-integer type to encode a bit-vector type $\mathcal{B}^w$. As a result, there might be some spare most-significant bits (MSBs) in an unsigned-integer variable. Normally, these bits have to be set to zeros (namely, the computation result is modulo 2^w) after each operation to guarantee the precision. Later in Sect. 5.1, we discuss an enhancement to perform modulo operations lazily only when needed, instead of applying it eagerly after each operator. Such laziness helps to generate shorter C programs. In the evaluation, we will also compare the effects of these two translation schemes. Next, we follow the order of Table 1 in the Btor2 paper [2] to introduce the Btor2 operators and their implementations in C.

4.4.1 Indexed Operators

Unsigned- and signed-extension operators `uext` and `sext` are implemented by type casting during variable assignment. The bit-slicing operator `slice` is implemented by first right-shifting the number of sliced least-significant bits and masking the spare MSBs to zeros.

[3] We stick to the ISO C17/C18 standard [9]; GNU C offers an `unsigned __int128` type, but not every software analyzer supports it. Arbitrary-width integers are supported in ISO C23 [69], which can help further simplify the translation.

4.4.2 Unary Operators

The bitwise negation operator `not` is implemented by its counterpart `~` in C. The arithmetic operators `inc`, `dec`, and `neg` are implemented using the `++`, `--`, and `-` operators in C. The reduction operator `redand` (resp. `redor`) is implemented by comparing the operand to $2^w - 1$ (resp. 0) for an operand of type $\mathcal{B}^w$. As there is no native support in C to compute the sum of all bits modulo 2 (parity) in an integer variable, the reduction operator `redxor` is implemented by repeatedly shifting and XOR-ing the variable with itself, such that the result will end up in the least-significant bit.

4.4.3 Binary Operators

For bit-vectors, the (in)equality operators `eq`, `neq`, `gt`, `gte`, `lt`, and `lte` are implemented by the corresponding C operators. For arrays, the equality operator is implemented by looping the two arrays to find a different element. Bitwise operators `and`, `or`, and `xor`[4] and arithmetic operators `add`, `mul`, `div`, `rem` (remainder), and `sub` are all supported in C and can be directly implemented using the respective C operators. In the Btor2 language, the result of division by zero is defined to be the maximum number of the dividend's sort. Our translation takes this specification into account to generate equivalent C programs. Otherwise, division by zero is an undefined behavior in C.

Shifting operators `sll` (logical left shift) and `srl` (logical right shift) are implemented by the left- and right-shifting operators in C, respectively. According to the ISO C18 standard [9], the result of right-shifting a negative value is implementation-defined. Therefore, to ensure the intended behavior of the arithmetic right-shift operator `sra`, we always pad ones directly to the resulting value if the given operand is negative (i.e., MSB equals 1). In this way, we do not have to assume any specific implementation of the software verifiers.

Concatenating and rotating operators `concat`, `rol` (rotating left), and `ror` (rotating right), are not natively supported in C. We implement them by shifting and bitwise disjunction. For example, in order to concatenate node n_1 of type $\mathcal{B}^3$ and node n_2 of type $\mathcal{B}^5$, we use `var_1 << 5 | var_2`, assuming `var_1` and `var_2` are of type `unsigned char`.

The `read` operator for array types, which takes an array and an index, is implemented using C's array-access operators `[]`.

[4] The operators `nand`, `nor`, and `xnor` are implemented with the bitwise NOT operator.

4.4.4 Ternary Operators

The if-then-else operator `ite` works both for bit-vectors and arrays. It is implemented by the ternary operator `exp1 ? exp2 : exp3` in C.

The `write` operator takes an array, an index for where to write, an element for what to write, and returns an updated array. It is implemented using the standard syntax in C to modify the content of an array.

In a Btor2 circuit, a `write` operation essentially creates a new copy of the original array with an updated element. The original array is not modified in place, because other lines may still refer to it. In principle, if no lines access the original array after a `write` operation, the array can be updated in place without allocating a new array. In Sect. 5.2, we will discuss how to reduce unnecessary duplication of arrays.

4.5 Discussion

4.5.1 Correctness of Btor2C

As will be seen in Sect. 6, the correctness of Btor2C is empirically confirmed over a large benchmark set: Software verifiers obtain consistent answers on almost all translated C programs as hardware verifiers. The inconsistent results have been investigated and were caused by the unsoundness of verifiers.

Furthermore, many software verifiers, e.g., those in SV-COMP, produce *verification witnesses* [70] to explain their verdicts. In a recent publication, we translated software verification witnesses back to Btor2 witnesses with a C-to-Btor witness translator and presented a certifying hardware-verification framework Btor2-Cert [71] using software verifiers. The witnesses produced by Btor2-Cert can be validated by the Btor2 witness validator Btor2-Val, further fostering the reliability of the translation process in Btor2C.

4.5.2 Limitations

The current version of Btor2C does not support the translation of fairness constraints (keyword `fair`), liveness properties (keyword `justice`), and overflow detection (keywords `addo`, `divo`, `mulo`, and `subo`). It only supports one-dimensional arrays, whose indices and elements are both of bit-vector sorts. It does not handle external inputs of array sorts. These unsupported keywords and features do not appear in the collected benchmark set of more than a thousand Btor2 circuits.

4.6 Blasting Arrays into Bit-Vectors

Reasoning about arrays could be challenging for software analyzers. We can get rid of arrays in a Btor2 circuit completely by "blasting" them into bit-vectors, i.e., representing individual elements in an array with bit-vectors. The blasting procedure takes a Btor2 circuit with arrays as input and generates a behaviorally equivalent Btor2 circuit with only bit-vectors as output. The operations over an array are simulated by corresponding operations over the blasted bit-vectors: `read` and `write` operations are encoded by sequences of `ite` operations to return or update the corresponding bit-vector of the given index; `eq` and `neq` operations are encoded by element-wise comparison between two sequences of blasted bit-vectors.

While blasting produces 2^w bit-vectors for an array with index type $\mathcal{B}^w$, it allows Btor2AIGER [24] to further bit-blast a circuit with arrays into the AIGER format such that powerful bit-level model checkers become applicable to Btor2 circuits with arrays, and software verifiers may perform better because of the absence of arrays. In our evaluation, every Btor2 circuit in the collected benchmark set can be blasted within a second; The bit-level model checker ABC can solve many blasted circuits, and the software verifier CPAchecker can solve more blasted circuits than the original ones.

5 Enhancing the Translation Procedure

This section presents three attempts to further enhance the translation procedure of Btor2C.

5.1 Applying Modulo Operations Lazily

Some operators can work correctly without precise values of the operands, which offers an opportunity to apply modulo operations lazily in a translated program. For instance, consider the addition operator. If $a_1 \equiv a_2 \pmod{n}$ and $b_1 \equiv b_2 \pmod{n}$, we conclude that $a_1 + b_1 \equiv a_2 + b_2 \pmod{n}$ according to modular arithmetics. In other words, the addition operator does not need precise operands and works correctly for modular numbers (i.e., equivalence classes modulo n). By contrast, other operators might yield different results for modular numbers. For example, $a + kn > b$ does not guarantee $a > b$ when $k > 0$. Therefore, performing the modulo operation to the result of an operator is only necessary when the result is used in another operator that requires precise operand values. A similar approach [72] has been explored in SMT solving for fixed-width bit-vectors, where modulo operations are

applied lazily when translating bit-vector formulas to integer formulas (also known as *int-blasting* [73]).

Btor2C provides an option for the lazy application of modulo operations. If the option is enabled, Btor2C analyzes whether the precise value is required for each node by looking at the node's fan-outs. If any of its fan-outs needs the precise computation result of the node, the modulo operation will be applied to it. Otherwise, the modulo operation will be skipped, and the result could be a modular number of the precise value. Operators that require precise operand values mainly include inequalities as well as indices for reading and writing arrays. As an example, if we enable *lazy modulo* to translate the Btor2 circuit in Fig. 2a, the modulo operations in lines 12 and 14 of the program in Fig. 2b can be omitted, because `input_5` and `var_6` are used only in addition and subtraction, which do not need precise operand values.

5.2 Reducing Unnecessary Duplication of Arrays

As discussed in Sect. 4.4.4, a `write` operation to an array creates a copy of the original array. Since reasoning about array variables is challenging for software verifiers, it is desirable to reduce unnecessary duplication of arrays in the translation process. One common pattern in Btor2 circuits is the combination of an `ite` operation and an `write` operation to conditionally update an array. We can avoid unnecessary duplication in this pattern as follows. Consider the example:

```
<nid1> write <sid> <array> <index> <value>
<nid2> ite   <sid> <cond>  <nid1>  <array>
```

The `ite` node conditionally selects between the original array and the written array. If the original array is not accessed after the `ite` operation, we can omit duplicating the array in the C program by translating this pattern into a ternary operation:

```
array[index] = cond ? value : array[index];.
```

In a similar vein, various transformation rules for `ite`, `read`, and `write` operations have been applied to SMT solving for bit-vectors and arrays [74] to increase sharing of subterms in the transformed formulas. By contrast, we aim to avoid unnecessary array copies during circuit-to-program translation.

5.3 Exploring Different Topological Orders

In a translated C program, intermediate signals of the Btor2 circuit need to be evaluated in a topological order from inputs to outputs. Exploring different topological orders offers a dimension to optimize translated C programs with respect to various objectives. For example, we can compute an optimal topological order to minimize array duplications incurred by `write` operations via integer linear programming (ILP). Given a Btor2 circuit with N intermediate signals, we formulate the ILP problem as follows:

1. For each intermediate signal with node ID i, create a variable x_i to encode its position in the optimal topological order and add a constraint $1 \leq x_i \leq N$.
2. For each intermediate signal with node ID i and any of its fan-in with node ID j, add a constraint $x_i > x_j$ to ensure the topological order.
3. For each `write` operation with node ID w, create a variable c_w to indicate when a copy of the written array is needed and add a constraint $0 \leq c_w \leq 1$. A copy is needed if and only if $c_w = 1$.
4. For each array that is both accessed and written, consider any pair of one of its access operations and one of its `write` operations. Let the node ID of the access operation be a and the node ID of the `write` operation be w. Add a constraint $x_w - x_a + N \times c_w \geq 0$ to require a copy of the written array if the access operation is scheduled after the `write` operation (if $x_w < x_a$, c_w must be 1 to satisfy the constraint).
5. Set the objective function to minimize the sum of all indicator variables c_w.

Since solving ILP problems is NP-complete, Btor2C implements a linear-time *as-late-as-possible* (ALAP) [75] scheduling for better scalability. Whenever an array access can be evaluated before a `write` operation to the array, the ALAP scheduling orders the access operation before the `write` operation to avoid duplicating the array. Consider the following sequence of array operations in Btor2:

```
3 sort array 2 1
...
7 state 3
8 write 3 7 6 4
9 read 1 7 5
```

Using the default topological order, Btor2C will create a copy of the array state at line 7 before the `write` operation at line 8:

```
SORT_3 var_8;
copy_sort_3(state_7, var_8);
var_8[var_6] = var_4;
SORT_1 var_9 = state_7[var_5];
```

With ALAP scheduling, Btor2C moves the `read` operation before `write`, eliminating the need for an extra array copy and allowing the original array to be reused:

```
// Read before write
SORT_1 var_9 = state_7[var_5];
```

```
// No array duplication, use pointer
SORT_1* var_8 = state_7;
var_8[var_6] = var_4;
```

Our evaluation shows that ALAP scheduling avoided array duplication in Btor2 tasks, leading to run-time reduction by more than 30 % for Cbmc and CPAchecker.

6 Evaluation

In this section, we will report the results from our extensive experiments on more than a thousand hardware-verification tasks, conducted to assess the Btor2-to-C translator Btor2C and the performance of software analyzers on translated hardware-verification tasks. Specifically, we aim to answer the following research questions in the evaluation:

RQ1: Is it feasible to translate Btor2 circuits to C programs via Btor2C?
RQ2: Are the proposed enhancement techniques for Btor2C effective?
RQ3: Can software analyzers solve the translated verification tasks?
RQ4: Can software analyzers complement hardware model checkers?
RQ5: Can blasting arrays to bit-vectors aid bit-level model checkers and software analyzers in solving verification tasks?
RQ6: Is Btor2C more effective and robust than other translators from hardware to software?

6.1 Benchmark Set

We collected hardware-verification tasks in Btor2 format from various sources, including the benchmark suites used in the 2019 and 2020 Hardware Model Checking Competitions [3], and the explicit-state model-checking tasks derived from the BEEM project [76]. To test our enhancement techniques for array translation, we also handcrafted 84 Btor2 circuits with arrays. The entire benchmark set along with a complete list of sources are available in a public repository of word-level verification tasks.[5]

We excluded tasks with bit-vectors longer than 64 bits, because Btor2C cannot translate these tasks into standard ISO C18 programs. In this section, experiments were primarily conducted on the collected 1 498 Btor2 tasks. Among these tasks, 1 341 contain only bit-vector sorts (referred to as bit-vector or BV tasks), while the remaining 157 manipulate both bit-vector and array sorts (referred to as array tasks). The bit-vector category contains 476 unsafe tasks (with a known specification violation) and 865 safe tasks (for which the specification is satisfied). Within the array category, 20 tasks are unsafe and 137 are safe. To answer RQ2, an additional evaluation was conducted on the 84 handcrafted array tasks, comprising 48 unsafe and 36 safe tasks.

We translated all Btor2 circuits to C programs using the proposed tool Btor2C (commit `7cf65f0`), assuming the LP64 data model, and to LLVM-IR programs using Btor2MLIR [19] (commit `2e06dec`). The original Btor2 circuits, as well as the translated C programs and AIGER circuits, are available in the aforementioned online repository. We also contributed the translated C programs to the sv-benchmarks collection, the largest freely-available benchmark set in the community of software verification and testing.

Although Btor2AIGER does not directly support Btor2 circuits with array sorts, such circuits can be translated to equivalent Btor2 circuits with only bit-vectors by applying the blasting procedure described in Sect. 4.6, which can the be further bit-blasted to AIGER circuits.

For our benchmark set, translating a Btor2 circuit to a C program or an AIGER circuit, or blasting arrays in a Btor2 circuit to bit-vectors, took less than a second. As this time is negligible, we excluded translation overhead when measuring the run time of the compared tools. An input task with the required format is directly provided to each tool. To facilitate the comparison with v2c, we additionally gathered 22 C programs translated by v2c from its repository.[6]

6.2 State-of-the-Art Hardware and Software Analyzers

To adequately reflect the state of the art of hardware and software analysis, we evaluated the most competitive tools from HWMCC [4], SV-COMP [5], and Test-Comp [6]. A wide range of techniques implemented in these tools, including BMC [77], k-induction (KI) [78, 79], IC3/property-directed reachability (PDR) [80], predicate abstraction (PA) [54, 60], and symbolic execution (SE) [81], were investigated in our experiments.

6.2.1 Hardware Model Checkers

We selected the state-of-the-art bit-level model checker ABC [15] (commit `af1de4f`), and word-level model checkers AVR [14] (commit `bdbfc83`) and BtorMC [2] (commit `6603ed7`). AVR was the winner of HWMCC

[5] https://gitlab.com/sosy-lab/research/data/word-level-hwmc-benchmarks

[6] https://github.com/rajdeep87/verilog-c

2020, and BtorMC has a strong implementation of BMC and k-induction. ABC consumes AIGER circuits as input while AVR and BtorMC directly analyze Btor2 circuits. We evaluated the implementations of BMC and PDR in both ABC [82] and AVR [83] as well as the implementations of BMC and k-induction in BtorMC.

6.2.2 Software Analyzers

For verification of C programs, Cbmc [11], Esbmc [12], and CPAchecker [10] were selected because they performed well in the category *ReachSafety-Hardware* of SV-COMP 2024 [5], which consists of C programs translated from Btor2 circuits by Btor2C. We evaluated the BMC implementations of all three tools, as well as predicate abstraction [84] in CPAchecker, and k-induction in Cbmc and Esbmc. For testing, we used a famous symbolic-execution engine Klee [13] due to its strong performance in the category *ReachSafety-Hardware* of Test-Comp 2024. For the LLVM-IR programs translated by Btor2MLIR, we employed SeaHorn [16] (version `14.0.0-rc0-79a7cf21`) to generate verification conditions in constrained Horn clauses (CHCs), which were subsequently solved by Spacer [85], a PDR-based CHC solver. We took the versions of CPAchecker [86], Esbmc [87], and Klee in SV-COMP and Test-Comp 2024, obtained from their respective DOIs in an archiving repository for formal-methods tools.[7] Cbmc was sourced from its official release `5.95.1` and combined with the execution script used in SV-COMP. SeaHorn was extracted from its official Docker image `docker.io/seahorn/seahorn-llvm14:nightly`. In addition, we augmented the execution script of Cbmc to perform k-induction on the translated programs.

In the following discussion, we use ⟨*tool*⟩-⟨*algorithm*⟩ to denote the implementation of a specific algorithm in a particular tool. For example, BtorMC-KI refers to the k-induction implementation in BtorMC.

6.3 Experimental Setup

All experiments were conducted on a cluster of machines running Ubuntu 24.04 (64 bit), each equipped with a 3.4 GHz Intel Xeon E3-1230 v5 CPU (8 processing units) and 33 GB of RAM. Each task was limited to 2 CPU cores, 15 min of CPU time, and 15 GB of RAM. We used BenchExec [88, 89] to ensure reliable resource measurement and reproducible results. Since SeaHorn is not provided as a standalone executable and relies on third-party shared libraries, it was executed inside a Podman container[8] with the help of fm-weck [90].

[7] `https://gitlab.com/sosy-lab/benchmarking/fm-tools`

Table 1: Applying modulo operations eagerly vs. lazily

Trans. mode	Median #mod-ops	#Tasks solved by		
		Cbmc k-Ind.	CPAchecker Pred. Abs.	Esbmc k-Ind.
Eager	886	576	**279**	411
Lazy	**487**	**577**	270	**416**

6.4 Experimental Results

Below we present and discuss the results of our experiments designed to study the research questions.

RQ1: Feasibility of Btor2C. We applied Btor2C to 1 498 Btor2 tasks in our benchmark set, and it successfully translated all of the tasks to C programs. The translation time for each Btor2 circuit was below one second. The correctness of Btor2C, namely, whether it produces a behaviorally equivalent C program of an input circuit, is demonstrated empirically by the verdicts of the evaluated software analyzers. Table 3 shows that all four analyzers (Cbmc, Esbmc, CPAchecker, and Klee) did not produce any wrong result on the entire benchmark set.

We also evaluated Btor2C with the enhancement techniques in Sect. 5 enabled. The overhead of the enhancements was negligible, and the translation time of any Btor2 circuit in our benchmark set was still below one second. Therefore, we conclude that it is feasible to correctly translate Btor2 circuits to C programs with Btor2C. In RQ2, we will study the effect of the proposed enhancement techniques.

> Btor2C can translate all Btor2 circuits in our benchmark set, and the translation time for each circuit is below one second, demonstrating the feasibility to represent Btor2 circuits as C programs.

RQ2: Enhancement Techniques for Btor2C. Tables 1 and 2 show the effect of the proposed enhancement techniques. From Table 1, we observe that applying modulo operations lazily almost halved the number of such operations in C programs translated from the 1 341 bit-vector tasks. In terms of the number of solved tasks, the shortened C programs slightly boosted the performance of Cbmc and Esbmc's k-induction but had a negative

[8] The image `registry.gitlab.com/sosy-lab/research/data/word-level-hwmc-benchmarks/btor2mlir:2e06dec3` was used, which was built on top of SeaHorn's official Docker image.

Table 2: Effects of reducing array duplication

(a) On 84 handcrafted tasks

Translation mode	#Tasks reduc.	#Tasks solved by		
		CBMC k-Ind.	CPACHECKER Pred. Abs.	ESBMC k-Ind.
None	–	32	23	**42**
ALAP (§ 5.3)	36	33	**34**	**42**
ITE (§ 5.2)	48	33	22	36
ALAP + ITE	**84**	**34**	**34**	36

(b) On 157 array benchmark tasks

Translation mode	#Tasks reduc.	#Tasks solved by		
		CBMC k-Ind.	CPACHECKER Pred. Abs.	ESBMC k-Ind.
None	–	**152**	0	2
ALAP (§ 5.3)	1	150	0	2
ITE (§ 5.2)	156	149	0	2
ALAP + ITE	**157**	148	**1**	2

impact on CPACHECKER's predicate abstraction. However, when comparing the accumulated CPU time for tasks solvable under both encodings, lazy application of modulo operations reduced the run time of all three tools: 2.5 % for CBMC, 6.2 % for CPACHECKER, and 8.9 % for ESBMC.

To test the effectiveness of the proposed approaches to reduce array duplication, we first conducted an evaluation on the handcrafted tasks, each containing one to three `write` operations. The second column of Table 2a lists the number of instances where array duplication was reduced. Note that, with both enhancements enabled, array duplication in these tasks can be completely avoided. From Table 2a, we observe that CBMC and CPACHECKER favor the C programs without array duplication (row "ALAP + ITE"), whereas ESBMC performed better on tasks translated by BTOR2C without any enhancements (row "None"). When comparing the two encodings (rows "None" vs. "ALAP + ITE") in terms of the accumulated CPU time for tasks solvable under both, the run time of CBMC and CPACHECKER was drastically reduced by 38.0 % and 35.8 %, respectively. For ESBMC, the results were mixed: it spent 85.4 % less time deriving proofs, but 310.1 % more time finding alarms, leading to a worse run-time performance overall.

We further extended our evaluation to the 157 collected array benchmark tasks, with results summarized in Table 2b. The BTOR2 tasks in this set typically contain one to two `write` operations, most of which are followed by an `ite` operation (recall the pattern described in Sect. 5.2). Similar as before, our enhancements could entirely avoid the array duplications during translation. However, the translated C programs with arrays seemed out of reach for CPACHECKER and ESBMC, making it difficult to draw a conclusion for the effectiveness of the enhancements. When comparing the number of solved cases, the trends varied among the evaluated verifiers and differed from the observation on the handcrafted tasks: CBMC solved fewer instances when enhancements were enabled, whereas CPACHECKER, which previously solved none, managed to solve one instance when both enhancements for translating arrays were enabled.

While the impact of the proposed enhancement techniques on verification performance varied across verifiers, they successfully achieved their goals of reducing modulo operations and array duplication. These techniques offer more translation possibilities to cater different software-verification tools and algorithms. The results may also interest developers of software analyzers, helping them understand why certain encodings benefit their tools.

From now on, we enable all enhancement techniques of BTOR2C for the evaluation in the subsequent RQs.

> The proposed enhancement techniques achieved their goals to reduce modulo operations and array duplication, providing more translation possibilities to cater different software-verification algorithms.

RQ3: Applicability of SW Analyzers. The results of the evaluated hardware model checkers, software verifiers, and software tester are summarized in Table 3. Word-level hardware model checkers AVR and BTORMC ran on the entire BTOR2 benchmark set; bit-level hardware model checker ABC ran on the bit-blasted AIGER circuits of BTOR2 circuits with bit-vectors only; software verifiers CBMC, CPACHECKER, and ESBMC ran on all translated C programs, while software tester KLEE ran only on unsafe verification tasks for bug hunting.

In the bit-vector category, ABC solved the most tasks overall, followed by CBMC, which surprisingly outperformed the two native BTOR2 model checkers, AVR and BTORMC. Observe that software analyzers were good at finding bugs in the translated C programs: CBMC identified most correct alarms in the experiment, and ESBMC and KLEE also detected more bugs than AVR. By contrast, hardware model checkers were better at computing correctness proofs: CBMC-KI, the best software verifier for proving correctness in our evaluation, achieved fewer than half of the proofs on bit-vector tasks.

In the array category, CBMC solved the most tasks. Notably, it found twice as many alarms as BTORMC, the second-best bug hunter in this category. BTORMC identified the most correct proofs, while AVR also delivered a fair number of proofs but failed to report any alarms. The number of tasks solved by other software

Table 3: Summary of the results for hardware and software verifiers on 1 498 benchmark tasks (ABC and SeaHorn only ran on 1 341 bit-vector tasks)

Analyzer	ABC	AVR	BtorMC	Cbmc	CPAchecker	Esbmc	Klee	SeaHorn
Algorithm	PDR	PDR	k-Induction	k-Induction	Pred. Abs.	k-Induction	Sym. Ex.	PDR
Input format	AIGER	Btor2		C				LLVM-IR
Translated by	Btor2AIG	–		Btor2C				Btor2MLIR
Correct results	**872**	708	675	**725**	271	418	321	399
BV proofs	**529**	426	198	**208**	184	93	–	127
BV alarms	**343**	238	336	**369**	86	323	320	272
Array proofs	–	44	**132**	**128**	1	0	–	–
Array alarms	–	0	**9**	**20**	0	2	1	–
Wrong proofs	0	0	0	0	0	0	–	9
Wrong alarms	0	0	0	0	0	0	0	344
Errors & Unknown	469	790	823	773	1 227	1 080	175	589

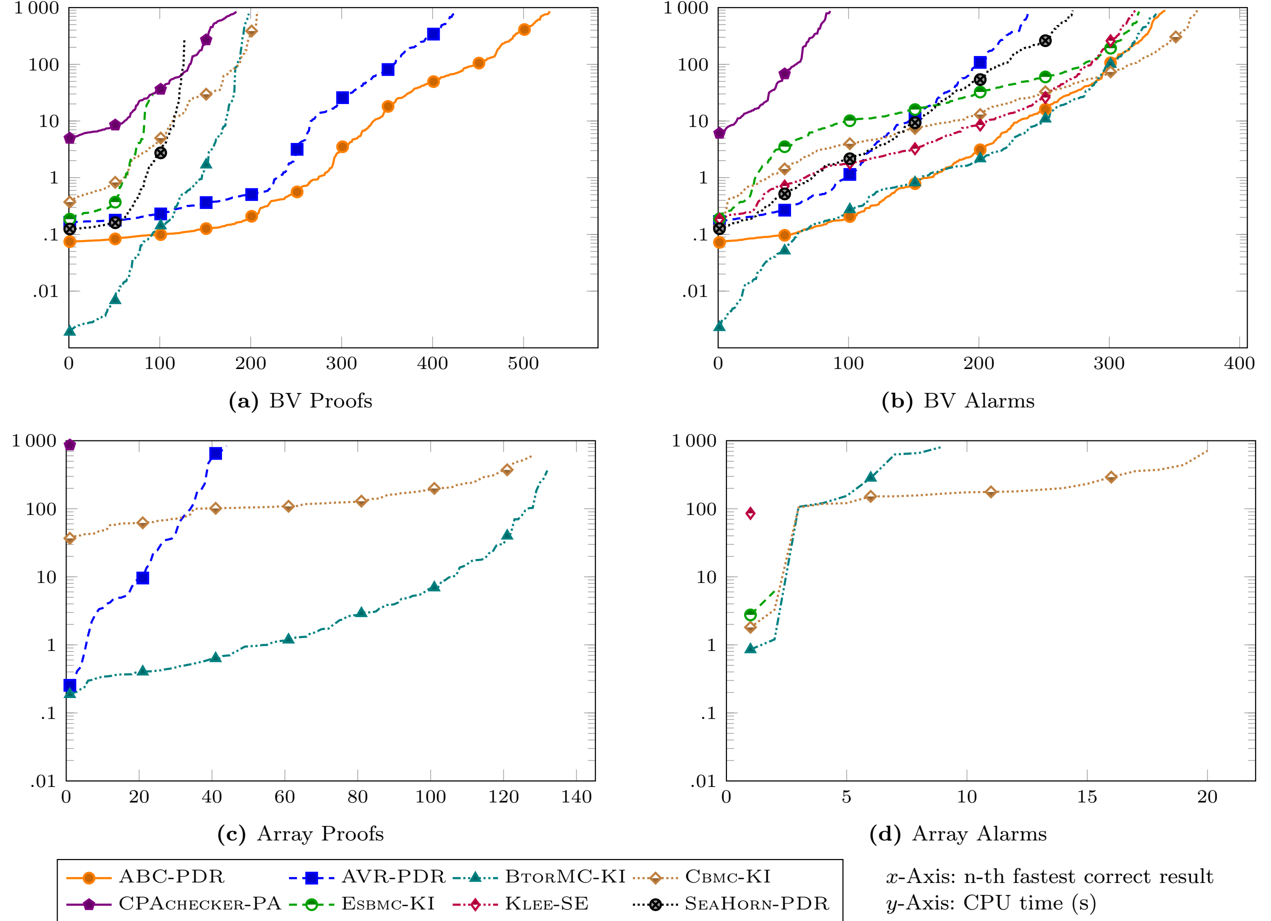


Fig. 4: Quantile plots for all correct proofs and alarms of verification tasks

analyzers was close to zero, indicating that they were incapable of handling these tasks effectively. Our results may inspire tool developers to investigate and mitigate the performance gap. The latest results of software verifiers and testers on a subset of the translated C programs can be found in the competition results of SV-COMP 2024 and Test-Comp 2024 under the category *ReachSafety-Hardware*.

The quantile plots of correct proofs and alarms are shown in Fig. 4. A data point (x, y) in the plots indicates that the respective tool correctly solved x tasks, each within y seconds of CPU time. For tasks with only bit-vectors, ABC was the most efficient and effective tool to

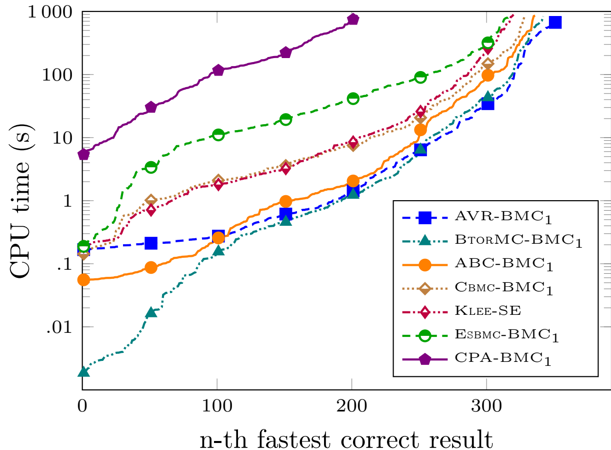


Fig. 5: Quantile plot comparing bug hunting on 476 bit-vector tasks (subscript refers to the number of unrolling steps in each iteration)

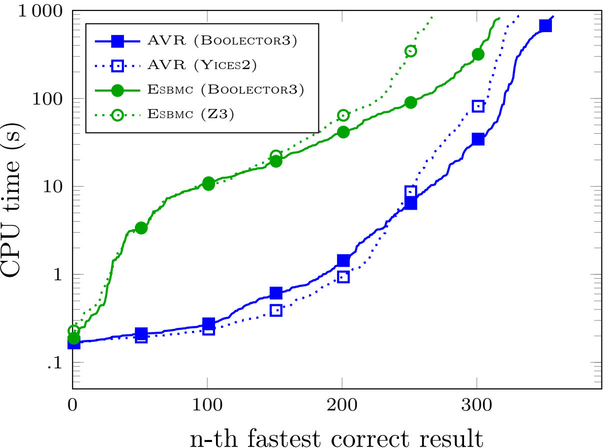


Fig. 6: Comparing BMC with different backend SMT solvers on 476 bit-vector tasks

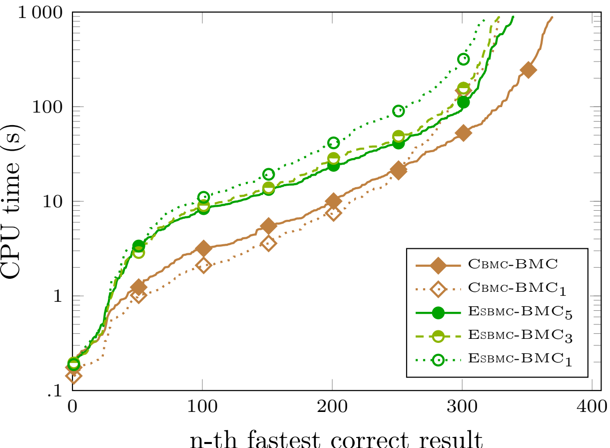


Fig. 7: Comparing BMC using different unrolling strategies on 476 bit-vector tasks (subscript refers to the number of unrolling steps in each iteration)

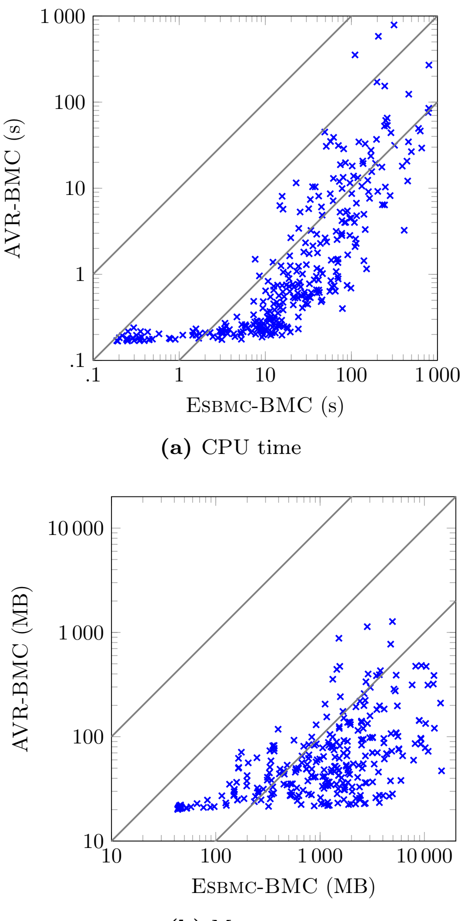
produce proofs, and CBMC was the best for bug hunting. While the numbers of alarms found by ESBMC and KLEE were more than AVR and close to ABC and BTORMC, they spent more time in finding bugs in general.

In our evaluation, we observe that PDR was the most competitive algorithm to find proofs for hardware model checkers, whereas software analyzers exhibited diverse strengths across different approaches. For instance, in the bit-vector category, CBMC's k-induction proved the property of 75 cases that CPACHECKER's predicate abstraction could not, while the latter succeeded in 51 cases where the former failed. Overall, software analyzers were able to deliver 1 proof and 28 alarms in bit-vector tasks, as well as 11 alarms in array tasks, that the hardware verifiers could not derive. Since software analyzers are good at finding property violations on our benchmark set, we further evaluate their ability to complement hardware model checkers for bug hunting in the next research question.

> The evaluated software verifiers and tester performed well on translated C programs and exhibited different strengths in finding proofs and alarms.

RQ4: Complementing HW Model Checkers. To assess the abilities of software analyzers to complement hardware model checkers for bug hunting, we compared the software verifiers' implementations of BMC, an algorithm designed for finding property violations, alongside KLEE, a well-known symbolic-execution tool for software testing, to the BMC implementations of hardware model checkers. Figure 5 presents the quantile plot of correct alarms for unsafe bit-vector tasks. The BMC implementations were configured to unroll one step per iteration across all tools. The performance of BMC implementations in CBMC and ESBMC closely matched those in hardware verifiers, with CBMC being the better among the two. The symbolic-execution engine in KLEE also performed decently. In total, software analyzers could identify 4 bugs that hardware verifiers were not able to locate within the resource limits.

To further investigate the factors behind efficient and effective bug hunting, we experimented with different satisfiability solvers and unrolling strategies whenever a tool allows flexible configuration through its user interface (e.g., via command-line options). Figure 6 shows the results of AVR using BOOLECTOR3 [2] and YICES2 [91] versus ESBMC using BOOLECTOR3 and Z3 [92]. The configurations with BOOLECTOR3 outperformed other SMT solvers, indicating that eagerly encoding SMT formulas to SAT formulas is more suitable to detect bugs in the



(b) Memory

Fig. 8: Usage of CPU time and memory of AVR-BMC and Esbmc-BMC (Both using Boolector3 and unrolling step by step)

evaluated Btor2 benchmark set. The observation is also confirmed by the results of ABC and Cbmc, both of which rely on SAT solving to check BMC queries.

Figure 7 shows the results of Cbmc and Esbmc with different unrolling strategies. Observe that, when Cbmc unrolled a program aggressively with multiple steps in each iteration (Cbmc-BMC, its default configuration in SV-COMP [93]), it found more bugs than unrolling the program step by step (Cbmc-BMC$_1$). Similar situation held for Esbmc: Unrolling with five steps in one iteration detected more bugs than unrolling with one or three steps. Therefore, we conclude that more aggressive unrolling is helpful for bug hunting.

Figure 8 further plots the CPU time and memory usage of the BMC implementations in AVR and Esbmc for finding correct alarms in the bit-vector category. Both tools are configured to employ the same backend SMT solver and unrolling strategy. A data point (x, y) in the scatter plots indicates the existence of a task correctly solved by both tools, for which Esbmc took x units of the computing resource and AVR took y units. AVR was often more efficient than Esbmc when both can find a bug and more effective overall. Nevertheless, Esbmc was able to detect bugs in 3 cases where AVR ran out of resources.

> Software analyzers are good at finding bugs: Cbmc, Esbmc, and Klee performed comparably to the state-of-the-art hardware model checkers ABC and AVR. Encoding BMC queries as SAT formulas and eager unrolling are helpful for finding bugs in Btor2 circuits.

RQ5: Blasting Arrays to Bit-Vectors. Table 4 compares the verification results of hardware model checkers and software analyzers on Btor2 circuits with arrays versus their blasted counterparts. Both CPAchecker and Klee were able to solve more tasks when arrays were eliminated. Moreover, array-to-bit-vector blasting enables ABC to analyze Btor2 circuits with arrays. We conclude that this blasting technique can be useful for tools that do not natively handle arrays effectively. However, it is important to note that the technique scales exponentially with the bit-width of the array index, meaning it is only feasible for arrays with small to medium index sizes. In our benchmark set, the maximum index width is 10.

> Blasting arrays to bit-vectors could be beneficial for certain tools. The resulting Btor2 circuits can be analyzed by bit-level model checker ABC and solved more effectively by Klee and CPAchecker.

RQ6: Comparison to v2c and Btor2MLIR. Btor2C is a lightweight tool, whose compiled binary is about 0.3 MB. By contrast, the precompiled v2c executable downloaded from its web archive[9] is 5.7 MB. While such difference is negligible given the capability of modern computers, we believe that a simple frontend language benefits tool implementation.

Besides implementation complexity, we also investigated the efficiency of the translation process. As mentioned in RQ1, Btor2C took less than a second to translate any Btor2 model in the benchmark set. Unfortunately, neither the v2c executable in the archive was runnable, nor was its source code compilable.[10] Therefore, we were not able to directly compare the translation efficiency of Btor2C and v2c.

[9] https://www.cs.ox.ac.uk/people/rajdeep.mukherjee/tacas16_v2c.tar.gz

[10] https://github.com/rajdeep87/verilog-c/issues/6

Table 4: Verification results on blasted array tasks

Analyzer Algorithm	ABC PDR	AVR PDR	BtorMC k-Induction	Cbmc k-Induction	CPAchecker Pred. Abs.	Esbmc k-Induction	Klee Sym. Ex.
Correct results (w/o blasting)	–	**44**	**141**	**148**	1	2	1
Correct results (w/ blasting)	**110**	41	137	143	**6**	2	**2**
Proofs	108	40	132	129	6	0	–
Alarms	2	1	5	14	0	2	2
Wrong results	0	0	0	0	0	0	0
Errors & Unknown	47	116	20	14	151	155	18

Table 5: Results for 22 programs generated by Btor2C and v2c

Analyzer Algorithm	CPAchecker Pred. Abs.		Esbmc k-Induction	
Translated by	Btor2C	v2c	Btor2C	v2c
Correct results	15	11	16	16
proofs	13	8	11	12
alarms	2	3	5	4
Errors & Unknown	7	11	6	6

As an alternative, we collected 22 C programs from v2c's benchmark repository and manually adapted them to the rules of SV-COMP. The original Verilog circuits of these C programs were translated to Btor2 by Yosys and further translated by Btor2C into another set of C programs. We compared the performance of CPAchecker and Esbmc on these two sets of 22 verification tasks in Table 5. Observe that CPAchecker was able to solve four more tasks translated by Btor2C, showing that prepending Yosys to Btor2C is comparable to using Verilog as frontend.

Btor2MLIR [19] translates a Btor2 circuit to an LLVM-IR program, which can be analyzed by model checkers like SeaHorn [16]. Table 3 compares SeaHorn on LLVM-IR programs translated by Btor2MLIR against the aforementioned hardware and software analyzers. We did not include array tasks since the translated LLVM-IR programs are not supported by SeaHorn [19]. Observe that, while SeaHorn found more proofs than Esbmc and more alarms than CPAchecker, it also suffered from several wrong proofs and a large number of wrong alarms on the benchmark set. Whether the unsoundness came from incorrect translation or verification is still under investigation.[11] By contrast, software verifiers for C programs did not produce any wrong result, showing the robustness of the proposed Btor2-to-C translation and verification flow.

[11] https://github.com/jetafese/btor2mlir/issues/32

The combination of Yosys and Btor2C is as effective as v2c. Additionally, Btor2C, together with C-program verifiers, are more robust than Btor2MLIR with SeaHorn, as the latter combination produced many incorrect results (especially false alarms).

7 Conclusion

Assuring the correctness of computing systems is challenging yet imperative. Therefore, we should embrace every opportunity to analyze our systems by bridging the gaps caused by modeling languages used in different research communities. We implemented Btor2C, a lightweight and open-source translator from sequential Btor2 circuits to C programs to enable the application of off-the-shelf software analyzers to hardware designs. We conducted large-scale experiments including more than a thousand verification tasks. State-of-the-art bit-level and word-level hardware model checkers as well as software verifiers and testers were evaluated empirically. Thanks to the simplicity of the Btor2 language, software analyzers performed well on the translated programs and complemented the hardware model checkers by detecting more bugs and uniquely solving 29 and 11 tasks in the bit-vector and array categories in our experiments, respectively.

Our translator Btor2C demonstrates a new spectrum of analysis options to hardware developers and verification engineers. The translator also simplifies the construction of a new set of hardware analyzers, because any software analyzer can now be used to solve hardware-verification tasks, with Btor2C as preprocessing. Btor2C has been combined with witness translation and validation to form a certifying hardware model checker using software verifiers as backend [71]. As part of our ongoing work, we are exploring performance differences between hardware and software analyzers to unify verification knowledge and advance the state of the art. One active direction involves using machine-learning-based algorithm selection to choose the most

suitable verification strategy based on the circuit-level features [94, 95]. In addition, we plan to investigate an alternative translation approach based on customized parsing, i.e., implementing a Btor2 frontend in some software analyzer, and compare the integrated translation against the proposed modular translator Btor2C.

Data-Availability Statement. To enhance the verifiability and transparency of the results reported in this article, the used software, verification tasks, and raw experimental results are available in a reproduction package on Zenodo [96]. Additional information is also available at a supplementary webpage: https://www.sosy-lab.org/research/btor2c/.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY) and 536040111 (Bridge).

References

1. 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), LNCS 13994, pp. 152–172. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_12
2. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. CAV, LNCS 10981, pp. 587–595. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
3. Hardware model-checking competition (HWMCC). https://hwmcc.github.io/. Accessed: 2025-01-14
4. Biere, A., Froleyks, N., Preiner, M.: Hardware model checking competition 2024. In: Proc. FMCAD, pp. 7–7. TU Wien Academic Press (2024). https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_6
5. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3), LNCS 14572, pp. 299–329. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
6. Beyer, D.: Software testing: 5th comparative evaluation: Test-Comp 2023. In: Proc. FASE, LNCS 13991, pp. 309–323. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_17
7. IEEE standard for Verilog hardware description language (2006). https://doi.org/10.1109/IEEESTD.2006.99495
8. Wolf, C.: Yosys open synthesis suite. https://yosyshq.net/yosys/. Accessed: 2023-01-29
9. 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
10. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV, LNCS 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
11. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS, LNCS 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
12. 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
13. 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
14. Goel, A., Sakallah, K.: AVR: Abstractly verifying reachability. In: Proc. TACAS, LNCS 12078, pp. 413–422. Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_23
15. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proc. CAV, LNCS 6174, pp. 24–40. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_5
16. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV, LNCS 9206, pp. 343–361. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
17. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University (2011). https://doi.org/https://doi.org/10.35011/fmvtr.2011-2
18. 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
19. Tafese, J., Garcia-Contreras, I., Gurfinkel, A.: Btor2MLIR: A format and toolchain for hardware verification. In: Proc. FMCAD, pp. 55–63. TU Wien Academic Press (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_13
20. Greaves, D.J.: A Verilog to C compiler. In: Proc. RSP, pp. 122–127. IEEE (2000). https://doi.org/10.1109/IWRSP.2000.855208
21. Mukherjee, R., Tautschnig, M., Kröning, D.: v2c: A Verilog to C translator. In: Proc. TACAS, LNCS 9636, pp. 580–586. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_38
22. Snyder, W.: Verilator. https://www.veripool.org/verilator/. Accessed: 2023-01-29
23. Smrcka, A., Vojnar, T.: Verifying parametrised hardware designs via counter automata. In: Proc. HVC, LNCS 4899, pp. 51–68. Springer (2007). https://doi.org/10.1007/978-3-540-77966-7_8
24. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Source-code repository of Btor2, BtorMC, and Boolector 3.0. https://github.com/Boolector/btor2tools. Accessed: 2026-02-14
25. Beyer, D., Podelski, A.: Software model checking: 20 years and beyond. In: Principles of Systems Design, LNCS 13660, pp. 554–582. Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_27
26. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPAchecker 3.0: Tutorial and user guide. In: Proc. FM, LNCS 14934, pp. 543–570. Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_30
27. Mukherjee, R., Kröning, D., Melham, T.: Hardware verification using software analyzers. In: Proc. ISVLSI, pp. 7–12. IEEE (2015). https://doi.org/10.1109/ISVLSI.2015.107
28. Mukherjee, R., Schrammel, P., Kröning, D., Melham, T.: Unbounded safety verification for hardware using software analyzers. In: Proc. DATE, pp. 1152–1155. IEEE (2016). https://ieeexplore.ieee.org/document/7459484
29. 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 14624, pp. 26–46. Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_2

30. IEEE standard for VHDL language reference manual (2019). https://doi.org/10.1109/IEEESTD.2019.8938196
31. 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
32. 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
33. 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
34. 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
35. McMillan, K.L.: Symbolic Model Checking. Springer (1993). ISBN: 978-1-4615-3190-6. https://doi.org/10.1007/978-1-4615-3190-6
36. 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, LNCS 2404, pp. 359–364. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
37. 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, LNCS 8559, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
38. Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementations. In: Proc. CAV, LNCS 8559, pp. 106–113. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_7
39. 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, LNCS 2304, pp. 213–228. Springer (2002). https://doi.org/10.1007/3-540-45937-5_16
40. 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
41. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier (competition contribution). In: Proc. TACAS (3), LNCS 14572, pp. 365–370. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22
42. 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). https://ceur-ws.org/Vol-3185/extended9547.pdf
43. Beyer, D., Ernst, G., Jonáš, M., Lingsch-Rosenfeld, M.: SV-LIB 1.0: A standard exchange format for software-verification tasks. arXiv/CoRR **2511**(21509) (2025). https://doi.org/10.48550/arXiv.2511.21509
44. 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
45. The CIRCT project: Circuit IR compilers and tools. https://circt.llvm.org/. Accessed: 2024-05-14
46. Bak, S., Bogomolov, S., Johnson, T.T.: HYST: A source transformation and translation tool for hybrid automaton models. In: Proc. HSCC, pp. 128–133. ACM (2015). https://doi.org/10.1145/2728606.2728630
47. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proc. CAV, LNCS 6806, pp. 379–395. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_30
48. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Taylor model flowpipe construction for non-linear hybrid systems. In: Proc, RTSS, pp. 183–192. IEEE (2012). https://doi.org/10.1109/RTSS.2012.70
49. Gao, S., Kong, S., Clarke, E.M.: Satisfiability modulo ODEs. In: Proc. FMCAD, pp. 105–112. IEEE (2013). https://doi.org/10.1109/FMCAD.2013.6679398
50. Yu, E., Biere, A., Heljanko, K.: Progress in certifying hardware model-checking results. In: Proc. CAV, LNCS 12760, pp. 363–386. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_17
51. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
52. Beckert, B., Hähnle, R.: Reasoning and verification: State of the art and current trends. IEEE Intelligent Systems **29**(1), 20–29 (2014). https://doi.org/10.1109/MIS.2014.3
53. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
54. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV, LNCS 1254, pp. 72–83. Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
55. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proc. POPL, pp. 191–202. ACM (2002). https://doi.org/10.1145/503272.503291
56. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279
57. 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
58. 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
59. 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
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. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV, LNCS 4144, pp. 123–136. Springer (2006). https://doi.org/10.1007/11817963_14
62. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference, LNCS 5947, pp. 165–176. Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
63. Beyer, D., Petrenko, A.K.: Linux driver verification. In: Proc. ISoLA, LNCS 7610, pp. 1–6. Springer (2012). https://doi.org/10.1007/978-3-642-34032-1_1
64. 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
65. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Proc. IFM, LNCS 2999, pp. 1–20. Springer (2004). https://doi.org/10.1007/978-3-540-24756-2_1

66. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM, LNCS 9058, pp. 3–11. Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
67. Cook, B.: Formal reasoning about the security of Amazon Web Services. In: Proc. CAV (2), LNCS 10981, pp. 38–47. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
68. 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
69. ISO/IEC JTC 1/SC 22: ISO/IEC 9899:2024: Information technology — Programming Languages — C. International Organization for Standardization (2024). https://www.iso.org/standard/82075.html
70. 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
71. 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), LNCS 14572, pp. 129–149. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_7
72. Barth, M., Heizmann, M.: A bit-vector to integer translation with bv2nat and nat2bv. In: Proc. SMT Workshop (2024). https://ceur-ws.org/Vol-3725/short9.pdf
73. Zohar, Y., Irfan, A., Mann, M., Niemetz, A., Nötzli, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Bit-precise reasoning via int-blasting. In: Proc. VMCAI, LNCS 13182, pp. 496–518. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_24
74. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Proc. CAV, LNCS 4590, pp. 519–531. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_52
75. Walker, R.A., Chaudhuri, S.: Introduction to the scheduling problem. IEEE Des. Test Comput. **12**(2), 60–69 (1995). https://doi.org/10.1109/54.386007
76. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Proc. SPIN, LNCS 4595, pp. 263–267. Springer (2007). https://doi.org/10.1007/978-3-540-73370-6_17
77. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS, LNCS 1579, pp. 193–207. Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
78. Donaldson, A.F., Haller, L., Kröning, D., Rümmer, P.: Software verification using k-induction. In: Proc. SAS, LNCS 6887, pp. 351–368. Springer (2011). https://doi.org/10.1007/978-3-642-23702-7_26
79. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. FMCAD, LNCS 1954, pp. 127–144. Springer (2000). https://doi.org/10.1007/3-540-40922-X_8
80. Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI, LNCS 6538, pp. 70–87. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
81. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
82. 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
83. Goel, A., Sakallah, K.: Model checking of Verilog RTL using IC3 with syntax-guided abstraction. In: Proc. NFM, pp. 166–185. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_11
84. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
85. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: Proc. CAV, LNCS 8044, pp. 846–862. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_59
86. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPAchecker 2.3 with strategy selection (competition contribution). In: Proc. TACAS (3), LNCS 14572, pp. 359–364. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
87. Menezes, R., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M.R., Tihanyi, N., Korovin, K., Cordeiro, L.: ESBMC v7.4: Harnessing the power of intervals (competition contribution). In: Proc. TACAS (3), LNCS 14572, pp. 376–380. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_24
88. 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
89. Beyer, D., Chien, P.C., Jankola, M.: BenchCloud: A platform for scalable performance benchmarking. In: Proc. ASE, pp. 2386–2389. ACM (2024). https://doi.org/10.1145/3691620.3695358
90. Beyer, D., Wachowitz, H.: FM-Weck: Containerized execution of formal-methods tools. In: Proc. FM, LNCS 14934, pp. 39–47. Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_3
91. Dutertre, B.: Yices 2.2. In: Proc. CAV, LNCS 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
92. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS, LNCS 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
93. Kröning, D., Tautschnig, M.: Cbmc: C bounded model checker (competition contribution). In: Proc. TACAS, LNCS 8413, pp. 389–391. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
94. Lu, Z., Chien, P.C., Lee, N.Z., Ganesh, V.: Algorithm selection for word-level hardware model checking (student abstract). Proc. AAAI **39**(28), 29426–29427 (2025). https://doi.org/10.1609/aaai.v39i28.35275
95. Lu, Z., Chien, P.C., Lee, N.Z., Gurfinkel, A., Ganesh, V.: Btor2-Select: Machine learning based algorithm selection for hardware model checking. In: Proc. CAV, LNCS 15931, pp. 296–311. Springer (2025). https://doi.org/10.1007/978-3-031-98668-0_15
96. Ates, S., Beyer, D., Chien, P.C., Lee, N.Z.: Reproduction package for STTT article 'Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator'. Zenodo (2025). https://doi.org/10.5281/zenodo.16933839