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

Dirk Beyer, Po-Chun Chien, and Nian-Ze Lee

LMU Munich, Munich, Germany

Abstract. Across the broad research field concerned with the analysis of computational systems, research endeavors are often categorized by the respective models under investigation. Algorithms and tools are usually developed for a specific model, hindering their applications to similar problems originating from other computational systems. A prominent example of such a situation is the area of formal verification and testing for hardware and software systems. The two research communities share common theoretical foundations and solving methods, including satisfiability, interpolation, and abstraction refinement. Nevertheless, it is often demanding for one community to benefit from the advancements of the other, as analyzers typically assume a particular input format. To bridge the gap between the hardware and software analysis, we propose Btor2C, a translator from word-level sequential circuits to C programs. We choose the Btor2 language as the input format for its simplicity and bit-precise semantics. It can be deemed as an *intermediate representation* tailored for analysis. Given a Btor2 circuit, Btor2C generates a behaviorally equivalent program in the language C, supported by many static program analyzers. We demonstrate the use cases of Btor2C by translating the benchmark set from the Hardware Model Checking Competitions into C programs and analyze them by tools from the Intl. Competitions on Software Verification and Testing. Our results show that software analyzers can complement hardware verifiers for enhanced quality assurance: For example, the software verifier VeriAbs with Btor2C as preprocessor found more bugs than the best hardware verifiers ABC and AVR in our experiment.

Keywords: Hardware compilation · Word-level circuit · Intermediate representation · Formal verification · Testing · Btor2 · SMT · SAT

1 Introduction

Computational systems have become more and more ubiquitous in our daily life and manifest themselves in various contexts, including VLSI circuits, software programs, and cyber-physical systems. To construct reliable systems, quality assurance has become an indispensable research topic. Numerous endeavors have been invested for different computational systems. Because of the ever-increasing system complexity and applications in safety-critical missions, it is of vital importance to take advantage of all available solutions for different types of systems to guarantee the quality and correctness.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 152–172, 2023.
https://doi.org/10.1007/978-3-031-30820-8_12

Formal verification and testing are two active fields of research to analyze and assure the quality of computational systems. The former decides with mathematical rigorousness whether a system conforms to a specification. The latter aims at generating input patterns and executing a system on a test suite to observe irregular output responses. Studies for formal verification or testing usually focus on a specific computational model, especially a sequential circuit (hardware) or a program (software). Tool competitions are also established based on modeling languages for input instances, such as the language Btor2 [64] used in the Hardware Model Checking Competitions (HWMCC) [28, 29], or the language C assumed by the Competitions on Software Verification (SV-COMP) [11, 14] and Testing (Test-Comp) [12, 13]. Unfortunately, such distinction erects a barrier between the two closely related research communities.

1.1 Our Motivations and Contributions

For the hardware community to easily benefit from state-of-the-art software-analysis techniques, we aim at *developing a lightweight yet effective translation flow to bridge the gap between hardware and software analysis*. There have been several attempts [48, 62] to compile hardware designs into software, mostly using the language Verilog as the input format. 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 in the frontend design, we resort to the language Btor2 [64], proposed recently to model word-level sequential circuits. A suite Btor2Tools [63] of utility tools is also provided for conveniently parsing, simulating, and bit-blasting (to the bit-level format Aiger [26]) Btor2 circuits. We emphasize the following two benefits of using Btor2 as the translation frontend 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 the underlying satisfiability solvers. Second, Btor2 is the input format used in the HWMCC. Many hardware model checkers support this format, and a large collection of benchmarking tasks is available for empirical evaluation. In practice, a Verilog circuit can be translated to Btor2 via Yosys [70], an open-source Verilog synthesis tool. Therefore, using Btor2 as frontend does not restrict the applicability of the translation flow.

Having settled down the frontend choice, our next question is: Should we make software analyzers support Btor2, or should we implement a standalone translator that does the job for all tools? We take the latter approach such that any software analyzer (from 76 available [25]) can in principle be used for hardware analysis. 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. 5 via comparison with the Verilog-to-C translator v2c [62].

Once a handy translator is viable, we are enthusiastic about *empirically comparing hardware and software analyzers on a large scale*. Similar experiments have been carried out for bounded [60] and unbounded [61] formal verification on a

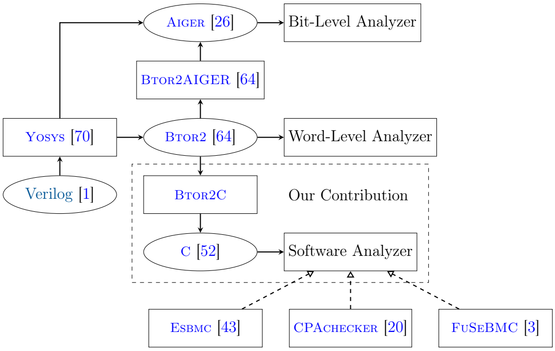


Fig. 1: Software analysis made readily available for hardware designs

small set of circuits. By building a translator on top of the Btor2 language, more than a thousand benchmarking tasks from the HWMCC are at our immediate disposal. To draw a more reliable conclusion on the performance comparison of state-of-the-art hardware and software analyzers, we evaluate bit-level and word-level hardware model checkers from HWMCC, software verifiers from SV-COMP, and software testers from Test-Comp, on the HWMCC benchmark set.

Our contributions in this paper are summarized below:

Novelty. (1) To bridge the gap between hardware and software analysis, we design and implement Btor2C, the first hardware-to-software compiler taking the format Btor2 [64] as input. Specifically, Btor2C accepts a Btor2 circuit and produces a behaviorally equivalent C program. Given a Verilog design, Btor2C (with the help of Yosys) makes off-the-shelf software verifiers and testers readily available for its analysis. In addition to bit-level and word-level analyzers, hardware developers will be equipped with more tool choices to perfect their designs, as shown in Fig. 1. (2) Btor2C makes it easy to construct new hardware analyzers by prepending the translator in front of any software analyzer. (3) Applying Btor2C to the HWMCC benchmark set, we submitted 1224 new tasks[1] to sv-benchmarks, the benchmark collection used by many researchers, including SV-COMP and Test-Comp. Developers of software analyzers can now assess their tools using the hardware-analysis counterparts as a new baseline.

Significance. (1) We conduct a large-scale evaluation involving hardware model checkers, software verifiers, and software testers on the HWMCC benchmark set. Our results show that software-analysis techniques can complement hardware model checkers. (2) The proposed lightweight translator makes software analyzers more accessible to the entire research community, as Btor2 can be used as an intermediate representation for analysis, not limited to hardware designs.

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

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

```
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 input_5;
10   SORT_1 state_3 = var_2;
11   for (;;) {
12     input_5 = nondet_uchar();
13     input_5 = input_5 & 0b111;
14     SORT_11 var_12 = state_3 == var_10;
15     SORT_11 bad_13 = var_12;
16     if (bad_13) {
17       ERROR: abort();
18     }
19     SORT_1 var_6 = state_3 + input_5;
20     var_6 = var_6 & 0b111;
21     SORT_1 var_8 = var_6 - var_7;
22     var_8 = var_8 & 0b111;
23     state_3 = var_8;
24   }
25 }
```

(b) C program (simplified for demo)

Fig. 2: An example Btor2 circuit (a) and its translated C program (b)

bit-vector is initialized to 0 (lines 2-4). In every iteration, the value of the bit-vector will be incremented by the value of the external input (lines 5-6) and then decremented by 1 (lines 7-8). The circuit reaches a bad state (i.e., violates the safety property) if the value of the bit-vector equals `0b111` (lines 12-13). The translated C program is shown in Fig. 2b. Btor2C first looks for the sorts used in the input Btor2 file. In this example, bit-vectors of 3 bits and 1 bit are used, and Btor2C encodes them with the shortest possible unsigned integer type `unsigned char` (lines 4-5). After sort declarations, Btor2C defines constants, declares inputs, and initializes circuit states (lines 6-10). An infinite loop is created to simulate the behavior of a sequential circuit. At the beginning of the loop, the safety property is evaluated. If the property is violated (namely, variable `bad_13` evaluates to *true*), the program reaches the error location at line 17. Otherwise, the next-state value (stored in variable `var_8`) is computed and assigned to the current state (lines 19-23), and another loop iteration follows. After the translation, we can apply software verifiers to the translated program in Fig. 2b to check whether the circuit in Fig. 2a conforms to the specified safety property.

2 Related Work

2.1 Compiling Hardware to Software

Several research efforts [48, 68] have been invested into representing a circuit as a program, whose primary goal is to accelerate hardware simulation. The most related work to ours is the Verilog-to-C translator v2c [62], used to translate hardware circuits into software programs for bounded [60] and unbounded [61] formal verification. Unlike v2c, our translator uses as frontend the Btor2 language, which is simple to parse and suitable for analysis. In Sect. 5, we compare the performance of software analyzers on C programs generated by v2c and our tool Btor2C.

2.2 Compiling Hardware to Intermediate Representation

Another line of research related to our work is the compilation of hardware to an intermediate representation that eases the burden of analysis. The motivation of these works is to interface real-world designs and problems described in a more abstract language with tools that use a primitive model representation. Our tool Btor2C shares a similar spirit because it interfaces problems in hardware analysis with software techniques. Among other tools, Verilog2SMV [51] and Ver2Smv [59] translate a Verilog circuit into SMV format [34, 56], which can be verified by tools like nuxmv [33]. QuteRTL [71] translates a register-transfer-level hardware design (usually in Verilog or VHDL) to Btor [31], an earlier version of Btor2. EBMC [55] generates SMT formulas in SMT-LIB 2 format [8], which encode the bounded model checking or k-induction problems of a Verilog circuit. Yosys [70], which translates a Verilog circuit into the Aiger or Btor2 formats, also serves the same purpose. Recently, there has been an interest to develop an intermediate language for the model-checking research community [67]. The project aims at providing an expressive frontend language as well as an efficient interface with backend model checkers.

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 [26]. The essential ingredients of Btor2 relevant to our discussion in Sect. 4 will be introduced below. For the complete syntax, please refer to the Btor2 publication [64].

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 bit-vector types $\mathcal{I}$ and $\mathcal{E}$, respectively, is denoted by $\mathcal{A}^{\mathcal{I}\rightarrow\mathcal{E}}$. 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, 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 general signature for a Btor2 operator is as follows: `<node id> <op> <sort id0> <node id1> [<node id2 [node id3]>]`, which defines a node to be the computation result of the operator `op` on node `id1` and optionally `id2` and `id3`. The result will have type `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 combinational circuit is in charge of the computation, and the memory elements store the circuit's state. The combinational circuit is a directed acyclic graph whose vertices are logic gates and edges are 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 `init`). 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 new values to be stored in the memory elements (namely, *next-state values*, described by `next`). At the end of the time frame, the next-state values are saved into the memory elements, which become the current state for 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 sequential circuit's combinational part, described by `bad`), 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 correctness witness.

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, unlike hardware, software model checking is in general undecidable, many research efforts have been invested into automated solutions to this problem [10, 19, 53], including predicate abstraction [5, 42, 47, 50], counterexample-guided abstraction refinement (CEGAR) [6, 36], and interpolation [49, 58]. The verification of industry-scale software such as operating-systems code [4, 7, 23, 32, 37, 54] is made feasible together by these solutions and the advances in SMT solving [9]. It is our research enthusiasm to explore how these concepts work on hardware.

4 Translating Btor2 to C

This section describes the proposed translator Btor2C[2], implemented in the language C with approximately 1600 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 C programming language 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 states. An endless loop is then used to mimic the state-transition behavior of the circuit throughout time frames: When a loop iteration begins, the safety property is evaluated over the current state and external input. If the property is violated, the program exits with an error. Otherwise, the next-state values are computed and stored into the state variables. This generic program reflects the reachability safety for sequential circuits.

```
void main() {
  // Define sorts and constants
  // Initialize states
  for (;;) {
    /* Evaluate safety property
    if (bad) {
      ERROR: abort();
    } */
    // Compute and assign next states
  }
}
```

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

The commented blocks in the generic program have to be replaced by C instructions to encode the concurrent computation of the sequential circuit. Btor2C assigns every node in the input Btor2 circuit a unique variable in the translated C program. 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 avoid irrelevant signals regarding model checking. Multiple `bad` keywords in a Btor2 file 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 variables in the translated C program. Suppose the unique ID of a line is `n`. If the line defines a sort, it is named `SORT_n` in the C file. If the line defines a state or an input, it is named `state_n` or `input_n`, respectively. If the line defines a node used for state initialization, transition, or property evaluation,

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

it is named `init_n`, `next_n`, or `bad_n`, respectively, to honor the keywords **init**, **next**, or **bad**. For the rest of the nodes, we name their variables `var_n` in the C file.

4.3 Expressing Btor2 Sorts in C

The language Btor2 supports two sorts: bit-vectors and arrays. Whenever possible, Btor2C represents a bit-vector type $\mathcal{B}^w$ 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, because no 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. 5, the state-of-the-art software verifiers and testers have a decent performance on the translated programs. In practice, only 20 % of the collected Btor2 benchmarking circuits have bit-vectors longer than 64 bits, so we consider the restriction acceptable.

For Btor2 arrays, Btor2C represents them by static arrays. Suppose the sort ID for an array type $\mathcal{A}^{\mathcal{I}\rightarrow\mathcal{E}}$ is **n**. Let its index type $\mathcal{I}$ be $\mathcal{B}^w$ and element type $\mathcal{E}$ be encoded by `SORT_m`. Then $\mathcal{A}^{\mathcal{I}\rightarrow\mathcal{E}}$ is encoded by the following C instruction: `typedef SORT_m SORT_n[1 << w];`, which means `SORT_n` is an array with 2^w objects 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. 4.5, we discuss the possibility of performing the modulo operation to results lazily only when needed, instead of applying it eagerly after each operator. Such laziness helps to generate shorter C programs and provides an opportunity for software analyzers to work more efficiently. 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 [64] to introduce the Btor2 operators and their implementations in C.

Indexed Operators. Unsigned- and signed-extension operators **uext** and **sext** can be implemented by type casting during the 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 C18 standard [52]; GNU C offers an `unsigned __int128` type, but not every software analyzer supports it. Recently, there is a proposal to support arbitrary-width integers in ISO C23, which will further simplify the translation.

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.

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 input 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 language Btor2, the result of division-by-zero is defined to be the maximum number of the operands' sort. Our translation takes this specification into account to generate equivalent C programs. Otherwise, division-by-zero would be considered as 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 [52], 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 implemented 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 simply implemented by C's syntax to access an array.

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.

Note that in a Btor2 file, a line with operator `write` essentially creates a new copy of the original array with one updated element. The original array is not replaced, because it might also be referred to by other lines. In principle, if no lines access the original array after a `write` operation, the operation could modify the element in place without allocating a new array. For now, Btor2C always copies a new array during a `write` operation for simplicity.

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

4.5 Applying Modulo Operations Lazily

Observe that there are some operators that can work correctly without precise operand values, which offers us the opportunity to apply modulo operations lazily and save some computations in translated programs. 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 arithmetic. 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 where the result is used in another operator that requires precise operand values.

Btor2C provides an option for the lazy application of modulo operations. If the option is turned on, 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 the lazy behavior to translate the Btor2 circuit in Fig. 2a, the modulo operations in line `13` and line `20` 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.

4.6 Discussion

Correctness of the Translation. As will be seen in Sect. 5, the reliability of Btor2C is empirically validated over a large input set: Most software verifiers obtain consistent answers on the translated C programs as the hardware verifiers. For Btor2 models that violate the safety property, the violation witness generated by software verifiers can be transformed to that of the original Btor2 circuit as a certificate of the translation process. The Btor2Tools utility suite offers a simulator to check the transformed witness against the Btor2 model.

Limitations. The current version of Btor2C has no support yet for the translation of fairness constraints (keyword `fair`), liveness properties (keyword `justice`), and overflow detection (keywords `addo`, `divo`, `mulo`, and `subo`). In our evaluation, only supported keywords appear in the collected Btor2 circuits.

5 Evaluation

We evaluate the claims presented in Sect. 1.1 using the following research questions:

- **RQ1**: How do software analyzers perform on hardware-verification tasks?
- **RQ2**: Can software analyzers complement hardware model checkers?
- **RQ3**: What is the effect of the optimization in Sect. 4.5 on the verification of the translated C programs?
- **RQ4**: How effective is the proposed translator Btor2C in comparison with the Verilog-to-C translator v2c [62]?

To answer the above research questions, we evaluated the state of the art of hardware and software analyzers over a large benchmark set consisting of more than thousand hardware-verification tasks.

5.1 Benchmark Set

We collected hardware-verification tasks in both Btor2 and Verilog formats from various sources, including the benchmark suites used in the 2019 and 2020 Hardware Model Checking Competitions [29] and the explicit-state model-checking tasks derived from the BEEM project [65]. The whole benchmark set as well as a complete list of sources are available in the reproduction artifact [16] of this paper. We also contributed a set of verification tasks to the sv-benchmarks collection, the largest freely available benchmark set of the verification and testing community.

As the proposed translator Btor2C uses Btor2 as frontend, we translated tasks in Verilog to Btor2 with Yosys [70]. An aggregate of 1912 Btor2 tasks were collected. We excluded 414 tasks with bit-vectors longer than 64 bits, because Btor2C cannot translate these tasks into standard ISO C18 programs. Out of the remaining 1498 Btor2 tasks, 1341 use only bit-vector sorts, and the remaining 157 tasks manipulate both bit-vector and array sorts. The bit-vector category contains 473 unsafe tasks (with a known specification violation) and 868 safe tasks (for which the specification is satisfied). The array category contains 17 unsafe and 140 safe tasks.

We translated the remaining 1498 Btor2 tasks into C programs by the proposed tool Btor2C (tag `tacas23-camera`), assuming the LP64 data model. The 1341 tasks in the bit-vector category are also translated to Aiger by the translator Btor2AIGER, which is provided in the Btor2Tools utility suite. The original Btor2 models as well as the translated C programs and Aiger circuits are available in the reproduction package [16] and online[5].

Unfortunately, Btor2AIGER does not translate Btor2 circuits with array sorts to Aiger. In our benchmark set, translating a Btor2 file to either a C program or an Aiger circuit took less than a second. Therefore, we ignore the translation time in the run-time of compared tools. An input task with the required format is directly given to each tool. To facilitate the comparison with v2c, we additionally gathered 22 C programs translated by v2c from its repository[6].

5.2 State-of-the-Art Hardware and Software Analysis

To adequately reflect the state of the art of hardware and software analysis, we evaluated the most competitive tools from the Hardware Model Checking Competitions and Competitions on Software Verification and Testing. A wide range of analysis techniques implemented in these tools were investigated in our experiment. Due to space limitation, Sect. 5.4 will show the best configuration of each tool on our benchmark set.

Hardware Model Checkers. For hardware analysis, we selected the state-of-the-art bit-level model checker ABC [30] (commit `a9237f5`[7]) and AVR [46] version 2.1,

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

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

[7] `https://github.com/berkeley-abc/abc`

a word-level hardware model checker that won HWMCC 2020. The former takes Aiger circuits as input, and the latter directly consumes Btor2 models. We evaluated the implementations of bounded model checking (BMC) [27] and property directed reachability (PDR) [41, 45] in both ABC and AVR. Interpolation-based model checking (IMC) [57] in ABC and k-induction (KI) [69] in AVR were also assessed.

Software Analyzers. For software verifiers, we enrolled the first, second, and fourth ranked verifiers VeriAbs [2], CPAchecker [20], and Esbmc [43] of category *ReachSafety* in SV-COMP 2022. The 3rd ranked verifier PeSCo [66] was omitted because it selects algorithms from the CPAchecker framework. All verifiers were downloaded from the archiving repository[8] of the competition. (For Esbmc, the performance of an earlier version in SV-COMP 2021 was better than the latest version on our benchmark set, so we used the older version instead.) We tried the implementations of loop abstraction (LA) [38] in VeriAbs; predicate abstraction (PA) [18, 50], Impact [24, 58], and IMC [21] in CPAchecker; BMC and KI [17, 18, 39, 44] in both CPAchecker and Esbmc.

For software testers, the overall winner FuSeBMC [3] of Test-Comp 2022, which implements fuzz testing (fuzzing), was picked. We also experimented with other testers from the competition, but they failed to generate test suites on our benchmark set. FuSeBMC was downloaded from the archiving repository[9] of the competition.

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

5.3 Experimental Setup

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. Each task was limited to 2 CPU cores, 15 min of CPU time, and 15 GB of RAM. We used BenchExec[10] [22] to ensure reliable resource measurement and reproducible results.

5.4 Results

RQ1: Solving HW-Verification Tasks with SW Analyzers. To study the performance of software analyzers on hardware-verification tasks, we compared the selected software tools against the state-of-the-art hardware model checkers. The results are summarized in Table 1.

Note that some software verifiers are good at finding bugs in these tasks. VeriAbs found most correct alarms in the experiment, and Esbmc also detected more bugs than AVR. By contrast, hardware model checkers were better at computing correctness proofs. Even the best software configuration CPAchecker-PA for proving correctness only achieved fewer than a half of the proofs for

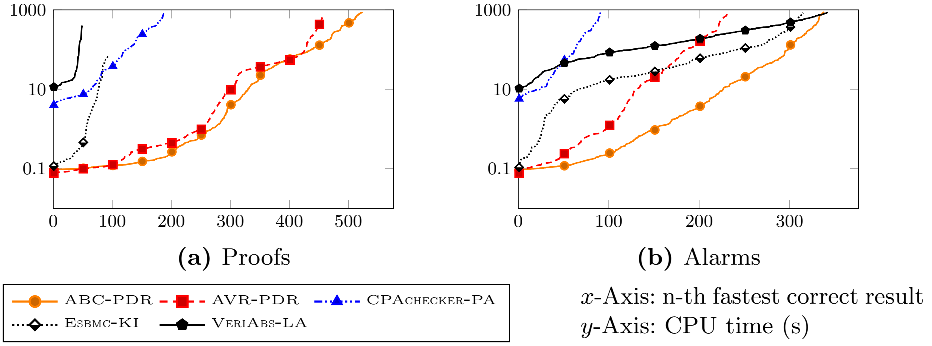
[8] https://gitlab.com/sosy-lab/sv-comp/archives-2022/-/tree/svcomp22

[9] https://gitlab.com/sosy-lab/test-comp/archives-2022/-/tree/testcomp22

[10] https://github.com/sosy-lab/benchexec

Table 1: Summary of the results for hardware and software verifiers (suffixes -*e* and -*l* stand for applying modulo operations eagerly or lazily, respectively)

Tool		ABC	AVR	CPAchecker		Esbmc		VeriAbs	
Algorithm		PDR	PDR	Pred. Abs.		*k*-Induction		Loop Abs.	
Input	Tasks	Aiger	Btor2	C-*e*	C-*l*	C-*e*	C-*l*	C-*e*	C-*l*
Correct results	1498	**862**	736	274	280	401	**410**	392	393
BV proofs	868	**524**	458	188	**189**	88	93	53	49
BV alarms	473	**338**	233	86	91	311	315	337	**342**
Array proofs	140	–	**45**	0	0	0	0	0	0
Array alarms	17	–	0	0	0	**2**	**2**	**2**	**2**
Wrong proofs		0	0	0	0	0	0	2	2
Wrong alarms		0	0	0	0	0	0	1	1
Timeouts		479	559	924	922	554	551	1049	1042
Out of memory		0	3	9	7	543	537	3	4
Other inconclusive		0	200	291	289	0	0	51	56



(a) Proofs

(b) Alarms

x-Axis: n-th fastest correct result
y-Axis: CPU time (s)

Fig. 4: Quantile plots for all correct proofs and alarms of bit-vector tasks

bit-vector tasks. In the array category, AVR delivered 45 correct proofs, whereas the software verifiers cannot solve any of them. Our results may inspire tool developers to investigate and alleviate the performance difference. Since we have contributed a category *ReachSafety-Hardware* of verification tasks to the common benchmark collection, the 2023 competition results of SV-COMP include evaluations of all participating tools on those new tasks.

The quantile plots of correct proofs and alarms for bit-vector tasks are shown in Fig. 4a and Fig. 4b, respectively. A data point (x, y) in the plots indicates that there are x tasks correctly solvable by the respective tool within a CPU time of y seconds. In our experiments, ABC is the most efficient and effective tool in producing proofs, and VeriAbs is the best for bug hunting. While the number of alarms found by Esbmc is more than AVR and close to ABC, it spent more time in finding bugs in general.

In our evaluation, we observe that PDR is the most competitive algorithm for both hardware model checkers, whereas software verifiers show diverse strengths in different approaches. To account for the difference in algorithms, we also compare implementations of the same algorithm in various analyzers.

BMC is one of the most popular formal approaches to detect errors. It is implemented by most of the evaluated tools. Software testers are also able to

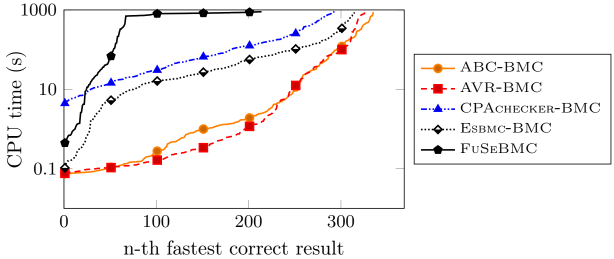


Fig. 5: Quantile plot comparing bug hunting (with BMC) on bit-vector tasks

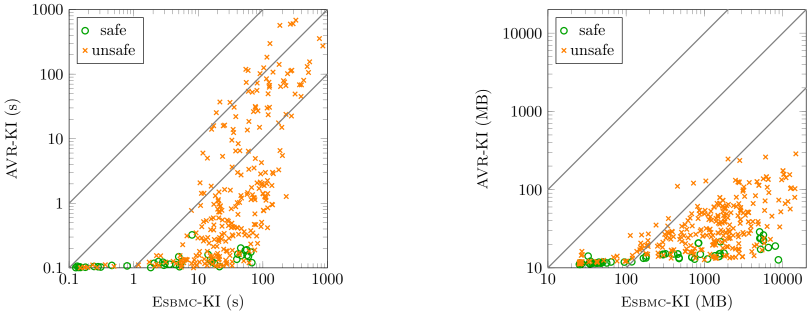


Fig. 6: CPU time (left) and memory (right) consumption of AVR-KI and Esbmc-KI

hunt bugs, and hence we include FuSeBMC, a derivative of Esbmc that combines BMC and fuzzing, into the comparison. Figure 5 shows the quantile plot of correct alarms for unsafe bit-vector tasks. Note that the performance of BMC implementations in software verifiers are close to those in hardware verifiers. However, FuSeBMC performed not as well as other competitors, indicating that fuzzing might not be fruitful for our benchmark set.

We also performed a head-to-head comparison of the k-induction implementations in AVR and Esbmc over the bit-vector and array tasks. Both tools rely on SMT solving for formula reasoning, so the confounding variables are fewer than other combinations. Figure 6 shows the scatter plots for the CPU time and memory usage of AVR and Esbmc to produce correct results. A data point (x, y) in the 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, but the latter solved 13 tasks that the former cannot solve.

RQ2: Complementing HW Model Checkers with SW Analyzers. Overall, hardware model checkers performed better than software analyzers on our benchmark set, which is expected since they have been heavily optimized for hardware-verification tasks. However, comparing the results of the tools for Table 1, we observed 43 tasks that were uniquely solved by software verifiers. Interestingly, 39 of these uniquely solved tasks have a violated property. Combining BMC with loop unwinding heuristics, e.g., the technique implemented in VeriAbs [2], is helpful to find bugs in these tasks. This phenomenon demonstrates that software-

Table 2: Results for 22 programs generated by Btor2C and v2c

Tool	CPAchecker		Esbmc		VeriAbs	
Algorithm	Pred. Abs.		k-Induction		Loop Abs.	
translated by	Btor2C	v2c	Btor2C	v2c	Btor2C	v2c
Correct results	15	11	16	13	12	7
proofs	13	8	11	11	7	3
alarms	2	3	5	2	5	4
Wrong results	0	0	0	1	0	0
Errors & Unknown	7	11	6	8	10	15

analysis techniques are able to complement hardware model checkers, which is facilitated by the proposed Btor2C translator. Some potential reasons affecting the effectiveness and efficiency of software analyzers will be discussed in Sect. 5.5.

RQ3: Optimization in Btor2C. Section 4.5 presented an optimization technique that performs modulo operations to intermediate results lazily, in order to generate shorter C programs. To assess whether this technique benefits the downstream software analysis, we compared the performance of the selected software verifiers, CPAchecker, Esbmc, and VeriAbs, on C programs translated by Btor2C with or without this optimization (namely, applying modulo operations lazily or eagerly, respectively).

The results of the best-performing algorithm for each tool in terms of the number of correct answers are summarized in Table 1, whose right panel also shows the results of the verifiers on these 2 sets of C programs. (CPAchecker-BMC actually solved more tasks than CPAchecker-PA, but it was mainly for bug hunting. Therefore, we reported the second best configuration, predicate abstraction, for CPAchecker.) If modulo operations are applied lazily instead of eagerly, the numbers of overall correct results are increased by roughly 2.2 % for both CPAchecker and Esbmc, and by 0.3 % for VeriAbs. Although VeriAbs found 4 fewer correct proofs if modulo operations are applied lazily, it reported 5 more correct alarms. Therefore, we conclude that generating shorter C programs by reducing modulo operations is an effective optimization in Btor2C. From now on, Btor2C enables this optimization by default.

RQ4: Comparison with v2c. Btor2C is a lightweight tool, whose compiled binary is smaller than 0.25 MB. By contrast, the precompiled v2c executable downloaded from its web archive[11] 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 Sect. 5.1, 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[12]. Therefore, we were not able to directly compare the translation efficiency of Btor2C and v2c.

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

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

As an alternative, we collected 22 C programs from v2c's benchmark repository and manually adapted them to the syntax rules used in 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 compare the performance of the evaluated software verifiers on these two sets of 22 verification tasks in Table 2. Observe that the three verifiers produced more correct results on the C programs generated by Btor2C, showing the benefit of using Yosys +Btor2 as frontend in the translation flow.

5.5 Discussion

From the experimental results shown above, we observe a notable performance difference between software and hardware analyzers. There are several possibilities to explain this outcome: First, the tasks were encoded in different formats for software and hardware analyzers. Btor2C encoded bit-vectors with unsigned integer types, which may contain some spare bits that complicate software analysis. Second, each analyzer uses a different backend logical solver. ABC encodes queries in propositional logic and uses SAT solving, while other tools resort to first-order formulas and SMT solving. (In our experiments, AVR used Yices2 [40], CPAchecker used MathSAT5 [35] for predicate abstraction and Boolector3 [64] for BMC, and Esbmc used Boolector3.) The ability of solvers may affect the analyzers' performance. Third, the internal modeling used by the analyzers varies. Software verifiers typically represent a program as a control-flow graph, which might be unnecessarily complex when the problem at hand is merely a state-transition system. Despite the above reasons, software verifiers were able to solve 43 tasks that the considered hardware model checkers cannot solve.

6 Conclusion

Assuring the correctness of computational systems is challenging yet imperative. Therefore, we should embrace every opportunity to analyze our systems by removing the barriers between research communities. We implemented the lightweight and open-source tool Btor2C for translating sequential Btor2 circuits to C programs, to enable the application of off-the-shelf software analyzers to hardware designs. We conducted a large-scale experiment including more than thousand verification tasks. State-of-the-art bit-level and word-level model checkers as well as software verifiers and testers were evaluated empirically. Thanks to the simplicity of the Btor2 language, software analyzers performed decently on the translated programs and complemented the hardware model checkers by detecting more bugs and uniquely solving 43 tasks in our experiment. 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. In the future, we wish to bridge the gap from the other direction. That is, we aim at translating programs into circuits and apply hardware analyzers to solve software problems.

Data-Availability Statement. To enhance the verifiability and transparency of the results reported in this paper, all used software, verification tasks, and raw experimental results are available in a supplemental reproduction package [16]. A previous version [15] of the reproduction package was reviewed by the Artifact Evaluation Committee. The updated version [16] fixes issues found by reviewers of the paper and the artifact. For convenient browsing of the data, interactive result tables are also available at https://www.sosy-lab.org/research/btor2c/.

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

Acknowledgements. We thank the SV-COMP community and an anonymous reviewer for pointing out the division-by-zero issue.

References

1. IEEE Standard for Verilog Hardware Description Language (2006). https://doi.org/10.1109/IEEESTD.2006.99495
2. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141 (2019). https://doi.org/10.1109/ASE.2019.00121
3. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. In: Proc. TAP. pp. 85–105. Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6
4. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Proc. IFM. pp. 1–20. LNCS 2999, Springer (2004). https://doi.org/10.1007/978-3-540-24756-2_1
5. 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
6. 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
7. 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
8. 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
9. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_11
10. 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
11. Beyer, D.: 11th Intl. Competition on Software Verification (SV-COMP 2022). https://sv-comp.sosy-lab.org/2022/, accessed: 2023-01-29
12. Beyer, D.: 4th Intl. Competition on Software Testing (Test-Comp 2022). https://test-comp.sosy-lab.org/2022/, accessed: 2023-01-29

13. Beyer, D.: Advances in automatic software testing: Test-Comp 2022. In: Proc. FASE. pp. 321–335. LNCS 13241, Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_18
14. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
15. Beyer, D., Chien, P.C., Lee, N.Z.: Reproduction package for TACAS 2023 submission 'Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator'. Zenodo (2022). https://doi.org/10.5281/zenodo.7303732
16. Beyer, D., Chien, P.C., Lee, N.Z.: Reproduction package for TACAS 2023 article 'Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator'. Zenodo (2023). https://doi.org/10.5281/zenodo.7551707
17. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
18. 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
19. 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
20. 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
21. 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
22. 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
23. Beyer, D., Petrenko, A.K.: Linux driver verification. In: Proc. ISoLA. pp. 1–6. LNCS 7610, Springer (2012). https://doi.org/10.1007/978-3-642-34032-1_1
24. Beyer, D., Wendler, P.: Algorithms for software model checking: Predicate abstraction vs. Impact. In: Proc. FMCAD. pp. 106–113. FMCAD (2012), https://www.sosy-lab.org/research/pub/2012-FMCAD.Algorithms_for_Software_Model_Checking.pdf
25. Beyer, D., Podelski, A.: Software model checking: 20 years and beyond. In: Principles of Systems Design. pp. 554–582. LNCS 13660, Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_27
26. 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
27. 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
28. 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
29. Biere, A., Froleyks, N., Preiner, M.: 11th Hardware Model Checking Competition (HWMCC 2020). http://fmv.jku.at/hwmcc20/, accessed: 2023-01-29

30. 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
31. Brummayer, R., Biere, A., Lonsing, F.: Btor: Bit-precise modelling of word-level problems for model checking. In: Proc. SMT/BPR. pp. 33–38. ACM (2008). https://doi.org/10.1145/1512464.1512472
32. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
33. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The NUXMV symbolic model checker. In: Proc. CAV. pp. 334–342. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
34. 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
35. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MATHSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
36. 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
37. Cook, B.: Formal reasoning about the security of Amazon web services. In: Proc. CAV (2). pp. 38–47. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
38. Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: Proc. DATE. pp. 1407–1412. IEEE (2015). https://doi.org/10.7873/DATE.2015.0245
39. Donaldson, A.F., Haller, L., Kröning, D., Rümmer, P.: Software verification using k-induction. In: Proc. SAS. pp. 351–368. LNCS 6887, Springer (2011). https://doi.org/10.1007/978-3-642-23702-7_26
40. Dutertre, B.: YICES 2.2. In: Proc. CAV. pp. 737–744. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
41. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proc. FMCAD. pp. 125–134. FMCAD Inc. (2011), http://dl.acm.org/citation.cfm?id=2157675
42. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proc. POPL. pp. 191–202. ACM (2002). https://doi.org/10.1145/503272.503291
43. 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
44. Gadelha, M.Y., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. Int. J. Softw. Tools Technol. Transf. **19**(1), 97–114 (February 2017). https://doi.org/10.1007/s10009-015-0407-9
45. 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

46. 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
47. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV. pp. 72–83. LNCS 1254, Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
48. Greaves, D.J.: A Verilog to C compiler. In: Proc. RSP. pp. 122–127. IEEE (2000). https://doi.org/10.1109/IWRSP.2000.855208
49. 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
50. 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
51. 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
52. 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
53. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
54. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
55. Kroening, D., Purandare, M.: EBMC. http://www.cprover.org/ebmc/, accessed: 2023-01-29
56. McMillan, K.L.: Symbolic Model Checking. Springer (1993). https://doi.org/10.1007/978-1-4615-3190-6
57. 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
58. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_14
59. 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
60. Mukherjee, R., Kroening, D., Melham, T.: Hardware verification using software analyzers. In: Proc. ISVLSI. pp. 7–12. IEEE (2015). https://doi.org/10.1109/ISVLSI.2015.107
61. Mukherjee, R., Schrammel, P., Kroening, 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
62. 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
63. 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
64. 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

65. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Proc. SPIN. pp. 263–267. LNCS 4595, Springer (2007). https://doi.org/10.1007/978-3-540-73370-6_17
66. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
67. Rozier, K.Y., Shankar, N., Tinelli, C., Vardi, M.: An open-source, state-of-the-art symbolic model-checking framework for the model-checking research community. https://www.aere.iastate.edu/modelchecker/, accessed: 2023-01-29
68. Snyder, W.: Verilator. https://www.veripool.org/verilator/, accessed: 2023-01-29
69. Wahl, T.: The k-induction principle (2013), http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf
70. Wolf, C.: Yosys open synthesis suite. https://yosyshq.net/yosys/, accessed: 2023-01-29
71. Yeh, H., Wu, C., Huang, C.R.: QuteRTL: Towards an open source framework for RTL design synthesis and verification. In: Proc. TACAS. pp. 377–391. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_26