FuzzBtor2: A Random Generator of Word-Level Model Checking Problems in BTOR2 Format*

Shengping Xiao[1], Chengyu Zhang[2], Jianwen Li[1]**, and Geguang Pu[1,3]**

[1] East China Normal University, Shanghai, China
spxiao@stu.ecnu.edu.cn, {jwli,ggpu}@sei.ecnu.edu.cn
[2] ETH Zurich, Zurich, Switzerland
chengyu.zhang@inf.ethz.ch
[3] Shanghai Trusted Industrial Control Platform Co., Ltd, Shanghai, China

Abstract. We present FuzzBtor2, a fuzzer to generate random word-level model checking problems in BTOR2 format. BTOR2 is one of the mainstream input formats for word-level hardware model checking and was used in the most recent hardware model checking competition. Compared to bit-level one, word-level model checking is a more complex research field at an earlier stage of development. Therefore, it is necessary to develop a tool that can produce a large number of test cases in BTOR2 format to test either existing or under-developed word-level model checkers. To evaluate the practicality of FuzzBtor2, we tested the state-of-the-art word-level model checkers AVR and Pono with the generated benchmarks. Experimental results show that both tools are buggy and not mature enough, which reflects the practical value of FuzzBtor2.

1 Introduction

Model checking plays an influential role in modern hardware design [4]. Its great success is inseparable from propositional methods such as Binary Decision Diagrams (BDDs) [10] and Boolean SATisfiability (SAT) solver [14]. Since BMC [6] was introduced, influential hardware model checking methods such as IMC [20], IC3 [9], and CAR [18] are all SAT-based. At the same time, many important efforts have been made to apply SAT-based model checking techniques to word-level verification tasks whose background theory are first-order logic [7,23,11,19,16]. These works all rely on more expressive reasoning techniques, i.e., Satisfiability Modulo Theories (SMT) [3] solvers. As the performance of the SMT solvers continues to improve [1,22], word-level hardware model checking has become a promising research area. Word-level reasoning is more powerful and opens up many possibilities for simplification [5]. It is strong evidence that a

* Jianwen Li is supported by National Natural Science Foundation of China (Grant #U21B2015 and #62002118) and Shanghai Pujiang Talent Plan (Grant #20PJ1403500). Geguang Pu is supported by National Key Research and Development Program (Grant #2020AAA0107800), and Shanghai Collaborative Innovation Center of Trusted Industry Internet Software.

** Corresponding authors.

word-level model checker, AVR [17], achieved the best results in the most recent hardware model checking competition [2].

Implementing word-level reasoning tools such as SMT solvers and word-level model checkers is much more complex and difficult than bit-level tools. For word-level model checking, which is a developing and immature area, it is an urgent requirement to obtain a large number of diverse benchmarks that can be used for bug finding and performance evaluation. Responding to this requirement, we present FuzzBtor2, a fuzzing tool that can generate random word-level model checking problems. We choose BTOR2 [21] as the format of output files, which is simple, line-based, and easy to parse. BTOR2 is also the current official format for the hardware model checking competition [2]. Most of mainstream word-level model checkers support BTOR2 format directly (AVR and Pono [19]) or indirectly (nuXmv [11] and IC3ia [13]). To evaluate whether FuzzBtor2 is practical, we test two state-of-the-art word-level model checkers AVR and Pono that can read BTOR2 files directly via BTOR2 files generated by FuzzBtor2, and generated test cases trigger various errors of both checkers. We expect that FuzzBtor2 becomes infrastructure for the development of word-level model checkers.

2 Word-Level Model Checking and BTOR2 Format

We assume that the reader is familiar with standard first-order logic terminology [3]. *Word*s generally refer to terms with bit-vector ranges, optionally combined with other theories. The background theory of BTOR2 is the Quantifier-Free theory of Bit Vectors with Arrays extension (QF_ABV), by which almost all computer system information can be encoded. And the invariant property is (one of) the most important property classes to verify.

A *model checking problem* consists of a transition system and a property to verify. A transition system is a tuple $S = (V, I, T)$ where

- V and V' are sets of variables in the present state and next state respectively;
- I is a set of formulas corresponding to the set of initial states;
- T is a set of formulas over $V \cup V'$ for the transition relation.

Given a transition system $S = (V, I, T)$, its state space is the set of possible variable assignments. I and T determine the reachable state space of S. The bad property is represented by a formula $\neg P$ over V. A model checking problem can be defined as follows: either prove that P holds for any reachable states of S, or disprove P by producing a counterexample. In the former, the system is *safe*, and in the latter, the system is *unsafe*. There are input variables in some transition systems, which can be modeled as state variables whose corresponding next states are unconstrained. Assume that a BTOR2 file includes n_s state variables, n_c constraints, and n_b bad properties. Its initial state space consists of n_s init-formulas. The transition relation consists of n_s next-formulas and n_c constraint-formulas. And the bad property consists of n_b bad-formulas. The sorts of init-formulas and next-formulas should be consistent with the corresponding state variables, and constraint-formulas and bad-formulas are Boolean sort.

3 The FuzzBtor2 Tool

FuzzBtor2 is an open-source software consisting of approximately 2400 lines of C++11 code. FuzzBtor2 does not rely on specific libraries and it is self-contained. In this section we introduce the usage and architecture of FuzzBtor2. The tool is available at https://github.com/CoriolisSP/FuzzBtor2.

3.1 Usage

The command to execute FuzzBtor2 in Linux systems is `./fuzzbtor [options]`. We present the usage and features of FuzzBtor2 along with the options here.

--seed INT This option is used to set the seed for the random number generator. Keeping other options, we could generate different test cases by changing the value of the random number seed. The default seed is 0.

--to-vmt Verification Modulo Theories (VMT) [12], which is an extension of SMT-LIB2 [3], is also used to represent symbolic transition systems and the properties to verify. vmt-tools [15] is a tool suite for VMT format, and it provides a translator from BTOR2 to VMT. However, vmt-tools supports only a subset of operators in BTOR2. By this option, the generated BTOR2 files only include the operators supported by vmt-tools, so that they can be translated into VMT format to test model checkers that take VMT files as input (e.g., IC3ia [13]).

--bv-states INT, --arr-states INT These options specify the numbers of bit-vector and array state variables. The default values are 2 and 0 respectively.

--max-inputs INT This option specifies the maximum number of input variables in the generated BTOR2 file. The actual number of input variables in the generated file may be smaller than the maximum. The default value is 1.

--bad-properties INT, --constraints INT These two options specify the numbers of bad properties and constraints in the generated BTOR2 file, and the default values are 1 and 0 respectively. The fuzzer currently does not support generating liveness properties and fairness constraints.

--max-depth INT A word-level model checking problem consisting of a transition system and properties to verify is essentially a set of first-order logic formulas. And formulas are represented by syntax trees in FuzzBtor2, so a word-level model checking problem corresponds to a set of syntax trees. This option specifies the maximum depth of these syntax trees. The default value is 4.

--candidate-sizes RANGE|SET FuzzBtor2 can get a set of positive integers from this option, which is used to specify sorts of variables. All sizes of indexes of array variables, elements of array variables, and sizes of bit-vector variables are in the set. The default set is $\{s \in \mathbb{Z} \mid 1 \leq s \leq 8\}$. Note that it does not allow to define a specific sort directly.

3.2 Architecture

The architecture of FuzzBtor2 consists of preprocessor, generator, and printer. Users of FuzzBtor2 only specify some arguments on the command line, and no other input is given. From command line arguments, the preprocessor sorts out

Algorithm 1: GenerateSyntaxTree

Input: A sort s of bit-vector or array, and a depth denoted by d
Output: A syntax tree of sort s with depth d

```
1  if d = 1 then
2      leafType := DecideLeafType()        // Decide the type of leaf node.
3      if leafType = constant then
4          return a constant
5      else if s ∈ candidateSort then
6          if leafType = input then
7              if there exists an input variable of sort s then
8                  return an existing input variable
9              if existInputNum < MaxInputNum then
10                 return an new input variable
11         else if leafType = state then
               // Similar to the case of input variables, omitted here.
12     return NULL                                  // Construction fails.
13 op := DecideOperator(s)
14 ⟨n, depths, sorts⟩ := DecideInformationOfSubtrees(op,d)
15 tree := NewTree(op)
16 for i = 1 ... n do
17     subTree := GenerateSyntaxTree(sorts[i], depths[i])        // Recursion.
18     if subTree = NULL then
19         return NULL
20     else
21         tree.AddSubTree(subTree)
22 return tree
```

the information required by the generator and saves it as a configuration. According to the configuration, the generator constructs some syntax trees that satisfy requirements of the number and sorts as stated in Sec. 2. These syntax trees encode a set of first-order logic formulas, which essentially is a model checking problem independent of the Btor2 format. At last, the printer outputs syntax trees constructed by the generator in Btor2 format.

The generator is the key component of FuzzBtor2. The generator constructs a syntax tree recursively, that is, a syntax tree with a depth greater than 1 consists of sub-syntax trees, operators, and some possible parameters (only for indexed operators). When the recursive process reaches the base case, i.e., a leaf node of the syntax tree, it randomly decides to return a (state or input) variable or a constant based on a certain probability. Due to the limitation of the number and sort of variables, if the generator chooses to return a variable, it may encounter a situation where the required leaf node cannot be constructed. Therefore, FuzzBtor2 does not guarantee that the Btor2 file can be successfully generated, and some parameters would cause the construction to fail. The overall process of constructing a syntax tree is described in Algorithm 1.

4 Experimental Evaluation

Tested Tools. In order to evaluate whether FuzzBtor2 is practical, we choose two state-of-the-art word-level model checkers AVR [17] and Pono [19] as tested tools. Both checkers can take BTOR2 as direct input format, and won the first and third place respectively in the 2020 Hardware Model Checking Competition [2].

Table 1: Overall results.

	Safe	Unsafe	Uniquely Solved	Error	Timeout
AVR (BV+ABV)	16 (11+5)	24 (11+13)	22 (13+9)	157 (78+79)	1 (0+1)
Pono (BV+ABV)	44 (20+24)	27 (13+14)	53 (24+29)	127 (67+60)	0

Table 2: Classification and statistics of error messages. The first type of error message of Pono has been confirmed by its developers.

	BV	ABV	Error Message
AVR	50	47	avr_word_netlist.cpp:912: static Inst* OpInst::create(OpInst::OpType, InstL, int, bool, Inst*, SORT): Assertion '0' failed.
	20	10	reach_y2.cpp:7367: void _y2::y2_API::inst2yices(Inst*, bool): Assertion '0' failed.
	1	3	reach_util.cpp:5785: void reach::Reach::check_correctness(): Assertion '0' failed.
	0	1	reach_y2.cpp:5365: virtual bool _y2::y2_API::get_assignment (Inst*, int&): Assertion 'e->get_sort_type() == bvtype' failed.
	2	3	reach_y2.cpp:7102: void _y2::y2_API::inst2yices(Inst*, bool): Assertion 'res != -1' failed.
	0	5	reach_y2.cpp:7113: void _y2::y2_API::inst2yices(Inst*, bool): Assertion 'res != -1' failed.
	1	3	Error: signal 11: build/bin/reach
	0	1	reach_y2.cpp:1784: void _y2::y2_API::add_gate_constraint (y2_expr &, y2_expr_ptr, std::string, Inst*, bool, bool): Assertion 'rhs != Y2_INVALID_EXPR' failed.
	0	1	reach_y2.cpp:6695: void _y2::y2_API::inst2yices(Inst*, bool): Assertion '0' failed.
	0	1	reach_y2.cpp:6002: y2_expr_ptr_y2::y2_API::create_y2_number (NumInst*): Assertion 'num->get_num() == 0' failed.
	4	3	reach_coi.cpp:943: bool reach::Reach::find_from_minset2 (Solver*, Inst*, InstS&, InstS&, std::set<std::__cxx11::basic_string<char> >&): Assertion 'ufType != "0"' failed.
	0	1	reach_util.cpp:5758: void reach::Reach::check_correctness(): Assertion '0' failed.
Pono	50	43	[boolector] boolector_slice: 'upper' must not be < 'lower'
	2	2	Segmentation fault (core dumped)
	7	7	free(): invalid pointer Aborted (core dumped)
	4	5	vector::_M_range_check: _n (which is 0) >= this->size() (which is 0)
	2	2	double free or corruption (out) Aborted (core dumped)
	2	1	[boolector] boolector_slice: 'upper' must not be >= width of 'exp'

Experimental Setups. We run FuzzBtor2 repeatedly with different parameters to generate a total of 200 test cases, in which 100 cases are array-free, i.e.,

without array variables (BV), and 100 cases include array variables (ABV). The command of FuzzBtor2 used for the former purpose is fuzzbtor2 --seed i --max-depth 4 --constraints 1 --bv-states 3 --arr-states 0 --max-inputs 3 --candidate-sizes 1..8. To generate BTOR2 models with array variables, the command is fuzzbtor2 --seed i --max-depth 4 --constraints 1 --bv-states 2 --arr-states 1 --max-inputs 3 --candidate-sizes 1..8. And i takes the value from 0 to 99. For every tested checker, the timeout to solve each instance is set to one hour.

Correctness. We use catbtor provided by btor2tools[4] [21] to verify the correctness of outputs of FuzzBtor2. All BTOR2 files generated by FuzzBtor2 pass the check of catbtor, which means all BTOR2 models generated by FuzzBtor2 are legal in syntax. Moreover, neither of the two tested tools (AVR or Pono) returns error messages that are relevant to the syntax issue of input BTOR2 files.

Results. We perform 200 calls to FuzzBtor2 and we get 100 BV test cases and 98 ABV test cases. Two calls for ABV test cases fail due to the situation discussed in sec. 3.2. The file sizes of the generated test cases are not large, with a maximum of 58 lines, a minimum of 22 lines, and an average of 39.2 lines. We use the generated 198 test cases to find bugs of AVR and Pono. All solving processes return results immediately, regardless of success or failure, except a situation where AVR timeouts on an ABV case. Table 1 presents overall statistical results. Neither AVR or Pono performs very well, since most of the test cases (157 vs. 127) trigger their bugs. And Table 2 presents the classification and statistics of error messages returned by tested tools. We encounter 12 and 6 different types of error messages for AVR and Pono respectively. It can be seen from Table 2 that ABV test cases trigger more types of errors than BV, which matches the fact that more code is covered in the process of solving a case in more complex theory. Considering both two tables, AVR performs worse than Pono in the experiments, where AVR solves fewer test cases and returns more types of error messages. Besides, the case where AVR timeouts is solved (Safe) by Pono, and is a BTOR2 file with only 43 lines, so we speculate that a performance issue occurs in AVR.

5 Conclusion

We have presented FuzzBtor2, an open-source tool for the generation of random BTOR2 files, by which the generated test cases can trigger various errors of state-of-the-art word-level model checkers. Several future works are being considered. First, if easy-to-trigger bugs of the tested tools are fixed, we could generate BTOR2 files of larger size and filter out benchmarks that can be used for performance evaluation through experiments. Second, there are some keywords (output, fair, and justice) of BTOR2 that are not supported by current FuzzBtor2, and we can extend the functionality of FuzzBtor2 to support them in future versions. Finally, as stated in sec. 3.2, the set of syntax trees constructed by the generator of FuzzBtor2 is essentially a model checking problem, independent of BTOR2 format. Therefore, it would be useful to print model checking problems randomly generated in other formats such as SMV [8] and VMT [12].

[4] https://github.com/boolector/btor2tools

Data-Availability Statement The artifact that supports the experimental results is available in Zenodo with the identifier https://doi.org/10.5281/zenodo.7234681 [24].

References

1. International satisfiability modulo theories competition, https://smt-comp.github.io/previous.html
2. Hardware model checking competition 2020 (2020), http://fmv.jku.at/hwmcc20/
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), www.SMT-LIB.org
4. Bernardini, A., Ecker, W., Schlichtmann, U.: Where formal verification can help in functional safety analysis. In: 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 1–8. ACM (2016)
5. Biere, A.: Tutorial on world-level model checking. In: 2020 Formal Methods in Computer Aided Design. IEEE, Haifa, Israel (2020)
6. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using sat procedures instead of bdds. In: Proceedings of the 36th annual ACM/IEEE Design Automation Conference. pp. 317–320 (1999)
7. Bjesse, P.: Word level bitwidth reduction for unbounded hardware model checking. Formal Methods in System Design **35**(1), 56–72 (2009)
8. Bozzano, M., Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: nuXmv 2.0. 0 user manual (2019)
9. Bradley, A.R.: Sat-based model checking without unrolling. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer (2011)
10. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on **100**, 677–691 (1986)
11. 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. 26th Int. Conf. on Computer Aided Verification. pp. 334–342. Springer, Vienna, Austria (2014)
12. Cimatti, A., Griggio, A., Tonetta, S.: The vmt-lib language and tools. arXiv preprint arXiv:2109.12821 (2021)
13. Daniel, J., Cimatti, A., Griggio, A., Tonetta, S., Mover, S.: Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In: Proc. 28th Int. Conf. on Computer Aided Verification. pp. 271–291. Springer (2016)
14. Eén, N., Sörensson, N.: An extensible sat-solver. In: International conference on theory and applications of satisfiability testing. pp. 502–518. Springer (2003)
15. Embedded Systems Unit, Digital Industry Center, Fondazione Bruno Kessler: vmt-tools (2022), http://es-static.fbk.eu/people/griggio/ic3ia/vmt-tools-latest.tar.gz
16. Goel, A., Sakallah, K.: Model checking of verilog rtl using ic3 with syntax-guided abstraction. In: NASA Formal Methods Symposium. pp. 166–185. Springer (2019)
17. Goel, A., Sakallah, K.: Avr: Abstractly verifying reachability. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 413–422. Springer (2020)
18. Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety model checking with complementary approximations. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 95–100. IEEE (2017)

19. Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.: Pono: a flexible and extensible smt-based model checker. In: Proc. 33th Int. Conf. on Computer Aided Verification. pp. 461–474. Springer (2021)
20. McMillan, K.L.: Interpolation and sat-based model checking. In: International Conference on Computer Aided Verification. pp. 1–13. Springer (2003)
21. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , btormc and boolector 3.0. In: Proc. 30th Int. Conf. on Computer Aided Verification. LNCS, vol. 10981, pp. 587–595. Springer, Oxford, UK (2018)
22. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The smt competition 2015–2018. Journal on Satisfiability, Boolean Modeling and Computation **11**(1), 221–259 (2019)
23. Welp, T., Kuehlmann, A.: Qf bv model checking with property directed reachability. In: 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 791–796. IEEE (2013)
24. Xiao, S.: Artifact – FuzzBtor2: A Random Generator of Word-Level Model Checking Problems in Btor2 Format (2022). https://doi.org/10.5281/zenodo.7234681