Btor2, BtorMC and Boolector 3.0

Aina Niemetz[1,2][0000−0003−2600−5283], Mathias Preiner[1,2][0000−0002−7142−6258], Clifford Wolf[3], and Armin Biere[1][0000−0001−7170−9242]

[1] Johannes Kepler University Linz
[2] Stanford University
[3] Symbiotic EDA

Abstract. We describe Btor2, a word-level model checking format for capturing models of hardware and potentially software in a bit-precise manner. This simple, line-based and easy to parse format can be seen as a sorted extension of the word-level format Btor. It uses design principles from the bit-level format Aiger and follows semantics of the Smt-Lib logics of bit-vectors with arrays. This intermediate format can be used in various verification flows and is perfectly suited to establish a word-level model checking competition. It is supported by our new open source model checker BtorMC, which is built on top of version 3.0 of our SMT solver Boolector. We further provide new word-level benchmarks on which these open source tools are evaluated.

Our format Btor2 generalizes and extends the Btor [5] format, which can be seen as a word-level generalization of the initial version of the bit-level format Aiger [2]. Btor is a format for quantifier-free formulas over bit-vectors and arrays with Smt-Lib [1] semantics but also provides sequential extensions for specifying word-level model checking problems with registers and memories. In contrast to Btor, which is tailored towards bit-vectors and one-dimensional bit-vector arrays, Btor2 has explicit sort declarations. It further allows to explicitly initialize registers and memories (instead of implicit initialization in Btor) and extends the set of sequential features with witnesses, invariant and fairness constraints, and liveness properties. All of these are word-level variants lifted from corresponding features in the latest Aiger format [4], the input format of the hardware model checking competition (HWMCC)[6, 3] since 2011. We provide an open source Btor2 tool suite, which includes a generic parser, random simulator and witness checker. We further implemented a reference bounded model checker BtorMC on top of our SMT solver Boolector. We consider Btor2 as an ideal candidate to establish a word-level hardware model checking competition.

Format Description

The syntax of Btor2 is shown in Figure 1. The sort keyword is used to define arbitrary bit-vector and array sorts. This not only allows to specify multi-dimensional arrays but can be extended to support (uninterpreted) functions,

Supported by Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

```
⟨num⟩      ::= positive unsigned integer (greater than zero)
⟨uint⟩     ::= unsigned integer (including zero)
⟨string⟩   ::= sequence of whitespace and printable characters without '\n'
⟨symbol⟩   ::= sequence of printable characters without '\n'
⟨comment⟩  ::= ';' ⟨string⟩
⟨nid⟩      ::= ⟨num⟩
⟨sid⟩      ::= ⟨num⟩
⟨const⟩    ::= 'const' ⟨sid⟩ [0-1]+
⟨constd⟩   ::= 'constd' ⟨sid⟩ ['-']⟨uint⟩
⟨consth⟩   ::= 'consth' ⟨sid⟩ [0-9a-fA-F]+
⟨input⟩    ::= ( 'input' | 'one' | 'ones' | 'zero' ) ⟨sid⟩ | ⟨const⟩ | ⟨constd⟩ | ⟨consth⟩
⟨state⟩    ::= 'state' ⟨sid⟩
⟨bitvec⟩   ::= 'bitvec' ⟨num⟩
⟨array⟩    ::= 'array' ⟨sid⟩ ⟨sid⟩
⟨node⟩     ::= ⟨sid⟩ 'sort' ( ⟨array⟩ | ⟨bitvec⟩ )
             | ⟨nid⟩ ( ⟨input⟩ | ⟨state⟩ )
             | ⟨nid⟩ ⟨opidx⟩ ⟨sid⟩ ⟨nid⟩ ⟨uint⟩ [⟨uint⟩]
             | ⟨nid⟩ ⟨op⟩ ⟨sid⟩ ⟨nid⟩ [⟨nid⟩ [⟨nid⟩]]
             | ⟨nid⟩ ( 'init' | 'next' ) ⟨sid⟩ ⟨nid⟩ ⟨nid⟩
             | ⟨nid⟩ ( 'bad' | 'constraint' | 'fair' | 'output' ) ⟨nid⟩
             | ⟨nid⟩ 'justice' ⟨num⟩ ( ⟨nid⟩ )+
⟨line⟩     ::= ⟨comment⟩ | ⟨node⟩ [ ⟨symbol⟩ ] [ ⟨comment⟩ ]
⟨btor⟩     ::= ( ⟨line⟩'\n' )+
```

Fig. 1: Syntax of BTOR2. Non-terminals ⟨opidx⟩ and ⟨op⟩ are indexed and non-indexed operators as defined in Table 1 (sequential part in red).

floating points and other sorts. As a consequence, BTOR2 is not backwards compatible with BTOR. For clarity, in Figure 1 we distinguish between node (line) identifiers ⟨nid⟩ and sort identifiers ⟨sid⟩, and do not allow an identifier to occur in both sets. Introducing sorts renders type specific keywords such as var, array and acond from BTOR obsolete. Instead, BTOR2 uses the keyword input to declare bit-vector and array variables of a given sort. Bit-vector constants are created as in BTOR with the keywords const[dh], one, ones and zero.

Bit-vector and array operators as supported by BTOR2 and their respective sorts are shown in Table 1. We use $\mathcal{B}^n$ for a bit-vector sort of width n, and $\mathcal{I}$ and $\mathcal{E}$ for the index and element sorts of an array sort $\mathcal{A}^{\mathcal{I}\to\mathcal{E}}$. Note that some bit-vector operators can be interpreted as *signed* or *unsigned*. In signed context, as in SMT-LIB, bit-vectors are represented in two's complement.

Sequential Extension

As shown in Figure 1, the sequential extension of BTOR2 introduces a state keyword, which allows to specify registers and memories. In contrast to BTOR, where registers are implicitly zero-initialized and memories are uninitialized, BTOR2 provides a keyword init to explicitly define initialization functions for states. This enables us to also model partial initialization. For example, initializing a memory with a bit-vector constant zero, zero-initializes the whole memory, whereas partially initializing a register can be achieved by applying a bit-mask to an uninitialized register.

Transition functions for both registers and memories are defined with the next keyword. It takes the current and next states as arguments. A state variable

indexed		
[su]ext w	(un)signed extension	$\mathcal{B}^n \to \mathcal{B}^{n+w}$
slice u l	extraction, $n > u \geq l$	$\mathcal{B}^n \to \mathcal{B}^{u-l+1}$
unary		
not	bit-wise	$\mathcal{B}^n \to \mathcal{B}^n$
inc, dec, neg	arithmetic	$\mathcal{B}^n \to \mathcal{B}^n$
redand, redor, redxor	reduction	$\mathcal{B}^n \to \mathcal{B}^1$
binary		
iff, implies	Boolean	$\mathcal{B}^1 \times \mathcal{B}^1 \to \mathcal{B}^1$
eq, neq	(dis)equality	$\mathcal{S} \times \mathcal{S} \to \mathcal{B}^1$
[su]gt, [su]gte, [su]lt, [su]lte	(un)signed inequality	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^1$
and, nand, nor, or, xnor, xor	bit-wise	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
rol, ror, sll, sra, srl	rotate, shift	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
add, mul, [su]div, smod, [su]rem, sub	arithmetic	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
[su]addo, [su]divo, [su]mulo, [su]subo	overflow	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^1$
concat	concatenation	$\mathcal{B}^n \times \mathcal{B}^m \to \mathcal{B}^{n+m}$
read	array read	$\mathcal{A}^{\mathcal{I}\to\mathcal{E}} \times \mathcal{I} \to \mathcal{E}$
ternary		
ite	conditional	$\mathcal{B}^1 \times \mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
write	array write	$\mathcal{A}^{\mathcal{I}\to\mathcal{E}} \times \mathcal{I} \times \mathcal{E} \to \mathcal{A}^{\mathcal{I}\to\mathcal{E}}$

Table 1: Operators supported by BTOR2, where $\mathcal{B}^n$ represents a bit-vector sort of size n and $\mathcal{A}^{\mathcal{I}\to\mathcal{E}}$ represents an array sort with index sort $\mathcal{I}$ and element sort $\mathcal{E}$.

without associated next function is treated as a *primary* input, i.e., it has the same behaviour as inputs defined via keyword input. Note that BTOR provides a next keyword for registers and an anext keyword for memories. Using sorts in BTOR2 avoids such sort specific keyword variants.

As in the latest version of AIGER [4], BTOR2 supports bad state properties, which are essentially negations of safety properties. Multiple properties can be specified by simply adding multiple bad state properties. Invariant constraints can be introduced via the constraint keyword and are assumed to hold globally. A witness for a bad state property is an initialized finite path, which reaches (actually, contains) a bad state and satisfies all invariant constraints.

Again as in AIGER [4], keywords fair and justice allow to specify (global) fairness constraints and (negations of) liveness properties. Each *justice* property consists of a set of Büchi conditions. A witness for a justice property is an infinite initialized path on which all Büchi conditions and all global fairness constraints are satisfied infinitely often. In addition, all global invariant constraints have to hold. The justice keyword takes a number (the number of Büchi conditions) and an arbitrary number of nodes (the Büchi conditions) as arguments.

Witness Format

The syntax of the BTOR2 witness format is shown in Figure 2. A BTOR2 witness consists of a sequence of valid input assignments grouped by (time) frames. It starts with 'sat' followed by a list of properties that are satisfied by the witness. A property is identified by a prefix 'b' (for **b**ad) and 'j' (for **j**ustice) followed by

a number i, which ranges over the number of defined *bad* and *justice* properties starting from 0. For example, 'b0 j0' refers to the first bad and first justice property in the order as they occur in the BTOR2 input. The list of properties is followed by a sequence of $k+1$ frames at time $t \in \{0, \ldots, k\}$. A *frame* is divided into a state and input part. The *state* part starts with '#t' and is mandatory for the first frame ($t = 0$) and optional for later frames ($t > 0$). It contains state assignments at time t. The *input* part starts with '@t' and consists of input assignments of the transition from time t to $t + 1$. If states are uninitialized (no init), their initial assignment is required to be specified in frame '#0'. The state part is usually omitted for $t > 0$ since state assignments can be computed from states and inputs at time $t - 1$. While don't care inputs can be omitted, our witness checker assumes that they are zero. Input and state assignments use the same numbering scheme as properties, i.e., states and inputs are numbered separately in the order they are defined, starting from 0. For example, 0 in frame '#t' (or '@t') refers to the first state (or input) as defined in the BTOR2 input. For justice properties we assume the witness to be lasso shaped, i.e., the next state, which can be computed from the last state and inputs at time k, is identical to one of the previous states at time $t = 0 \ldots k$. As in AIGER, a BTOR2 witness is terminated with '.' on a separate line.

⟨binary-string⟩	::=	[0-1]+
⟨bv-assignment⟩	::=	⟨binary-string⟩
⟨array-assignment⟩	::=	'[' ⟨binary-string⟩ ']' ⟨binary-string⟩
⟨assignment⟩	::=	⟨uint⟩ (⟨bv-assignment⟩ | ⟨array-assignment⟩) [⟨symbol⟩]
⟨model⟩	::=	(⟨comment⟩'\n' | ⟨assignment⟩'\n')+
⟨state part⟩	::=	'#' ⟨uint⟩ '\n' ⟨model⟩
⟨input part⟩	::=	'@' ⟨uint⟩ '\n' ⟨model⟩
⟨frame⟩	::=	[⟨state part⟩] ⟨input part⟩
⟨prop⟩	::=	('b' | 'j')⟨uint⟩
⟨header⟩	::=	'sat\n' (⟨prop⟩)+ '\n'
⟨witness⟩	::=	(⟨comment⟩'\n')+ | ⟨header⟩ (⟨frame⟩)+ '.'

Fig. 2: BTOR2 model and witness format syntax (sequential part in red).

Figure 3 illustrates a simple C program (left), the corresponding BTOR2 model with the negation of the assertion as a bad property (center), and a BTOR2 witness for the violated property (right). The BTOR2 model defines one bad property (`a == 3 && b == 3`), which is satisfied in frame 6. The corresponding witness identifies this property as bad property 'b0' (first bad property defined in the model). All states are initialized, hence '#0' is empty, and '@0' to '@6' indicate the assignments of input 0 (`turn`, the first input defined in the model) in frames 0 to 6, e.g., `turn` $= 1$ at $t = 0$, `turn` $= 0$ at $t = 1$ and so on. In frame 6, both states `a` and `b` reach value 3, and therefore property 'b0' is satisfied.

Tools

We provide a generic stand-alone parser for BTOR2, which features basic type checking and consists of approx. 1,500 lines of C code. We implemented a refer-

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
static bool read_bool () {
  int ch = getc (stdin);
  if (ch == '0') return false;
  if (ch == '1') return true;
  exit (0);
}
int main () {
  bool turn;             // input
  unsigned a = 0, b = 0; // states
  for (;;) {
    turn = read_bool ();
    assert (!(a == 3 && b == 3));
    if (turn) a = a + 1;
    else      b = b + 1;
  }
}
```

```
 1 sort bitvec 1
 2 sort bitvec 32
 3 input 1 turn
 4 state 2 a
 5 state 2 b
 6 zero 2
 7 init 2 4 6
 8 init 2 5 6
 9 one 2
10 add 2 4 9
11 add 2 5 9
12 ite 2 3 4 10
13 ite 2 -3 5 11
14 next 2 4 12
15 next 2 5 13
16 constd 2 3
17 eq 1 4 16
18 eq 1 5 16
19 and 1 17 18
20 bad 19
```

```
sat
b0
#0
@0
0 1 turn@0
@1
0 0 turn@1
@2
0 0 turn@2
@3
0 0 turn@3
@4
0 1 turn@4
@5
0 1 turn@5
@6
0 0 turn@6
```

Fig. 3: Example C program with corresponding BTOR2 model and witness.

ence bounded model checker BtorMC, which currently supports checking safety (aka. bad state) properties for models with registers and memories and produces witnesses for satisfiable properties. Unrolling the model is performed by symbolic simulation, i.e., symbolic substitution of current state expressions into next state functions, and incremental SMT solving. We also implemented a simulator for randomly simulating BTOR2 models. It further supports checking BTOR2 witnesses. The model checker is tightly integrated into our SMT solver Boolector [18], an award-winning SMT solver for the theory of fixed-size bit-vectors with arrays and uninterpreted functions. Since the last major version [18], we extended Boolector with several new features. Most notably, Boolector 3.0 now comes with support for quantified bit-vectors [24] and two different local search strategies for quantifier-free bit-vector formulas that don't rely on but can be combined with bit-blasting [22, 19, 21]. It further provides support for BTOR2. In contrast to previous versions of Boolector, Boolector 3.0 and all BTOR2 tools are released under the MIT open source license and the source code is hosted on GitHub[4].

Experiments

We collected ten real-world (System)Verilog designs with safety properties from various open source projects [28, 26, 27, 11]. The majority of these designs include memories. We used the open synthesis suite Yosys [29] to synthesize these designs into BTOR2 and SMT-LIB. For BTOR2, Yosys directly generates the models from a circuit description. For SMT-LIB, since the language does not support describing model checking problems, we used Yosys in combination with Yosys-SMTBMC to produce unrolled (incremental) problems.

We compared BtorMC against the most recent versions of Boolector (3.0) and Yices [10] (2.5.4), the two best solvers of the QF_ABV division of the

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

Benchmark	k	#bad	**BtorMC** Time[s]	**Boolector** Time[s]	**Yices** Time[s]
picorv32-check	30	23	**4.8**	18.9	10.8
picorv32-pcregs	20	3	**63.0**	293.0	TO
ponylink-slaveTXlen-sat	230	1	305.5	406.8	**145.6**
ponylink-slaveTXlen-unsat	231	1	183.8	131.4	**71.4**
VexRiscv-regch0-15	17	2	**9.6**	48.3	12.2
VexRiscv-regch0-20	22	2	528.8	**520.7**	2232.2
VexRiscv-regch0-30	32	2	TO	TO	TO
zipcpu-busdelay	100	50	**157.0**	287.0	181.2
zipcpu-pfcache	100	39	**17.4**	19.9	32.5
zipcpu-zipmmu	30	57	86.0	412.9	**46.5**

Table 2: BtorMC/Btor2 vs. unrolled Smt-Lib with a time limit of 3600 seconds, where k is the bound and #bad is the number of bad properties.

SMT competition 2017. The Btor2 models serve as input for BtorMC, and the incremental Smt-Lib benchmarks serve as input for Boolector and Yices. All benchmarks, synthesis scripts, generated files, log files and the source code of our tools for this evaluation are available at `http://fmv.jku.at/cav18-btor2`.

The results in Table 2 show that our flow using Btor2 as intermediate format is competetive with simple unrolling. Note that our model checker BtorMC issues incremental calls to Boolector. However, in Boolector, sophisticated word-level rewriting is currently disabled in incremental mode. We expect a major performance boost by fully supporting incremental word-level preprocessing.

Conclusion

We propose Btor2, a new word-level model-checking and witness format. For this format we provide a generic parser implementation, a simulator that also checks witnesses, and a reference bounded model checker BtorMC, which is tightly integrated with our SMT solver Boolector. These open source tools are evaluated on new real-world benchmarks, which we synthesized from open source hardware (System) Verilog models into Btor2 and Smt-Lib with Yosys. The tool Verilog2SMV [14] translates Verilog into model-checking problems in several formats, including nuXmv [7] and Btor. However, its translation to Btor is incomplete and development discontinued.

We plan to provide a translator from Btor2 into SALLY [25], and VMT [8], which are both extensions of Smt-Lib to model symbolic transition systems. It might also be interesting to translate incremental Smt-Lib benchmarks and horn clause models (as handled by, e.g., μZ [13]) into Btor2 and vice versa. We hope other compilers and model checkers such as SAL [9], EBMC [15] and ABC [16, 12] will provide support to produce and read Btor2 models. We want to extend the format to other logics, in particular to support lambdas as in [23]. There is also a need for fuzzing [20] and delta-debugging tools [17].

Last but not least, we want to use this format to bootstrap a word-level model checking competition, which of course needs more benchmarks.

References

1. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
2. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Tech. rep., FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2007)
3. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. p. 9. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102233, https://doi.org/10.23919/FMCAD.2017.8102233
4. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. rep., FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
5. Brummayer, R., Biere, A., Lonsing, F.: BTOR: Bit-precise modelling of word-level problems for model checking. In: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning. pp. 33–38. SMT '08/BPR '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1512464.1512472, http://doi.acm.org/10.1145/1512464.1512472
6. Cabodi, G., Loiacono, C., Palena, M., Pasini, P., Patti, D., Quer, S., Vendraminetto, D., Biere, A., Heljanko, K.: Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks. Journal on Satisfiability, Boolean Modeling and Computation **9**, 135–172 (2014 (published 2016))
7. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014)
8. Cimatti, A., Roveri, M., Griggio, A., Irfan, A.: Verification Modulo Theories. http://es.fbk.eu/projects/vmt-lib/
9. De Moura, L., Owre, S., Shankar, N.: The sal language manual. Computer Science Laboratory, SRI International, Menlo Park, Tech. Rep. CSL-01-01 (2003)
10. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (2014)
11. Gisselquist, D.: zipcpu. https://github.com/ZipCPU/zipcpu
12. Ho, Y., Mishchenko, A., Brayton, R.K.: Property directed reachability with word-level abstraction. In: FMCAD. pp. 132–139. IEEE (2017)
13. Hoder, K., Bjørner, N., de Moura, L.M.: μZ- an efficient engine for fixed points with constraints. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 457–462. Springer (2011)
14. Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: Verilog2smv: A tool for word-level verification. In: DATE. pp. 1156–1159. IEEE (2016)
15. Kroening, D.: Computing over-approximations with bounded model checking. Electr. Notes Theor. Comput. Sci. **144**(1), 79–92 (2006)
16. Long, J., Ray, S., Sterin, B., Mishchenko, A., Brayton, R.K.: Enhancing ABC for stabilization verification of systemverilog/vhdl models. In: DIFTS@FMCAD. CEUR Workshop Proceedings, vol. 832. CEUR-WS.org (2011)

17. Niemetz, A., Biere, A.: ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In: Bruttomesso, R., Griggio, A. (eds.) Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT 2013), affiliated with the 16th International Conference on Theory and Applications of Satisfiability Testing, SAT 2013, Helsinki, Finland, July 8-9, 2013. pp. 36–45 (2013)
18. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. JSAT **9**, 53–58 (2015)
19. Niemetz, A., Preiner, M., Biere, A.: Precise and complete propagation based local search for satisfiability modulo theories. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9779, pp. 199–217. Springer (2016)
20. Niemetz, A., Preiner, M., Biere, A.: Model-Based API Testing for SMT Solvers. In: Brain, M., Hadarean, L. (eds.) Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT 2017), affiliated with the 29th International Conference on Computer Aided Verification, CAV 2017, Heidelberg, Germany, July 24-28, 2017. p. 10 pages (2017)
21. Niemetz, A., Preiner, M., Biere, A.: Propagation based local search for bit-precise reasoning. Formal Methods in System Design **51**(3), 608–636 (2017). https://doi.org/10.1007/s10703-017-0295-6, https://doi.org/10.1007/s10703-017-0295-6
22. Niemetz, A., Preiner, M., Biere, A., Fröhlich, A.: Improving local search for bit-vector logics in SMT with path propagation. In: Proceedings of the Fourth International Workshop on Design and Implementation of Formal Tools and Systems, Austin, TX, USA, September 26-27, 2015. pp. 1–10 (2015)
23. Preiner, M., Niemetz, A., Biere, A.: Lemmas on demand for lambdas. In: DIFTS@FMCAD. CEUR Workshop Proceedings, vol. 1130. CEUR-WS.org (2013)
24. Preiner, M., Niemetz, A., Biere, A.: Counterexample-guided model synthesis. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10205, pp. 264–280 (2017). https://doi.org/10.1007/978-3-662-54577-5_15, https://doi.org/10.1007/978-3-662-54577-5_15
25. SRI International's Computer Science Laboratory: Sally - a model checker for infinite-state systems. https://github.com/SRI-CSL/sally
26. Wolf, C.: picorv32. https://github.com/cliffordwolf/picorv32
27. Wolf, C.: PonyLink. https://github.com/cliffordwolf/PonyLink
28. Wolf, C.: riscv-formal. https://github.com/cliffordwolf/riscv-formal
29. Wolf, C.: Yosys. https://github.com/YosysHQ/yosys