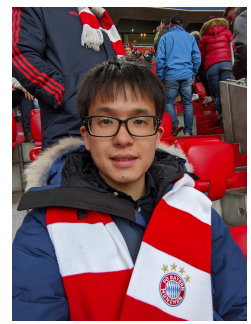
Bridging Hardware and Software Analysis with Btor2C: A Word-Level-Circuit-to-C Translator

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

LMU Munich, Germany

AVM 2023, 2023-09-13

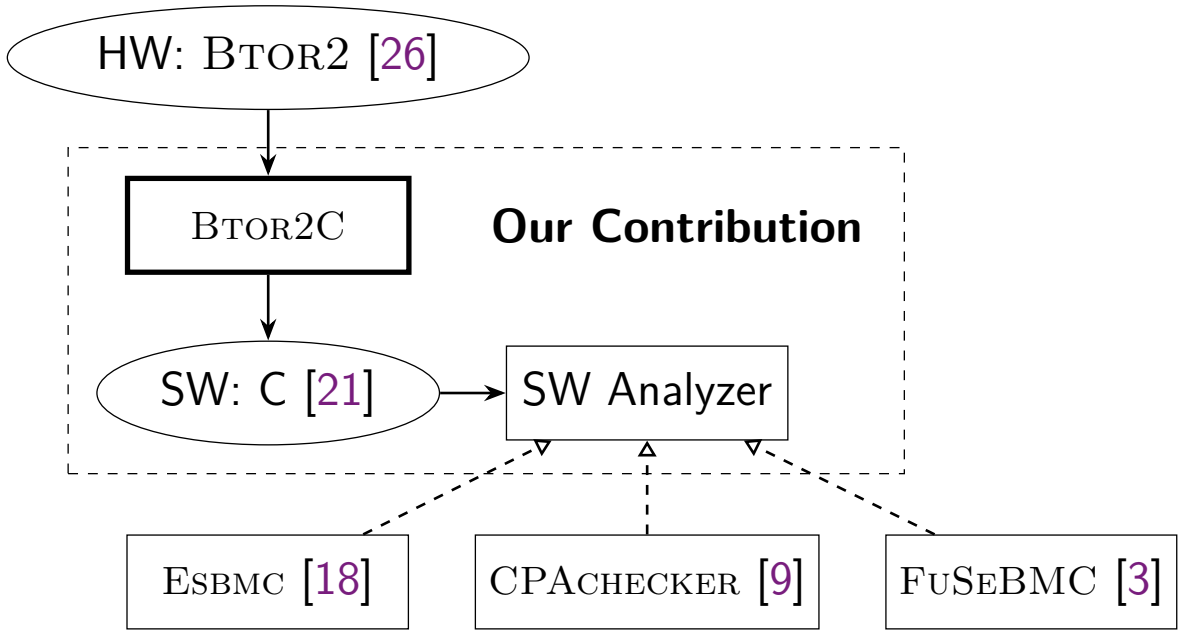




LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

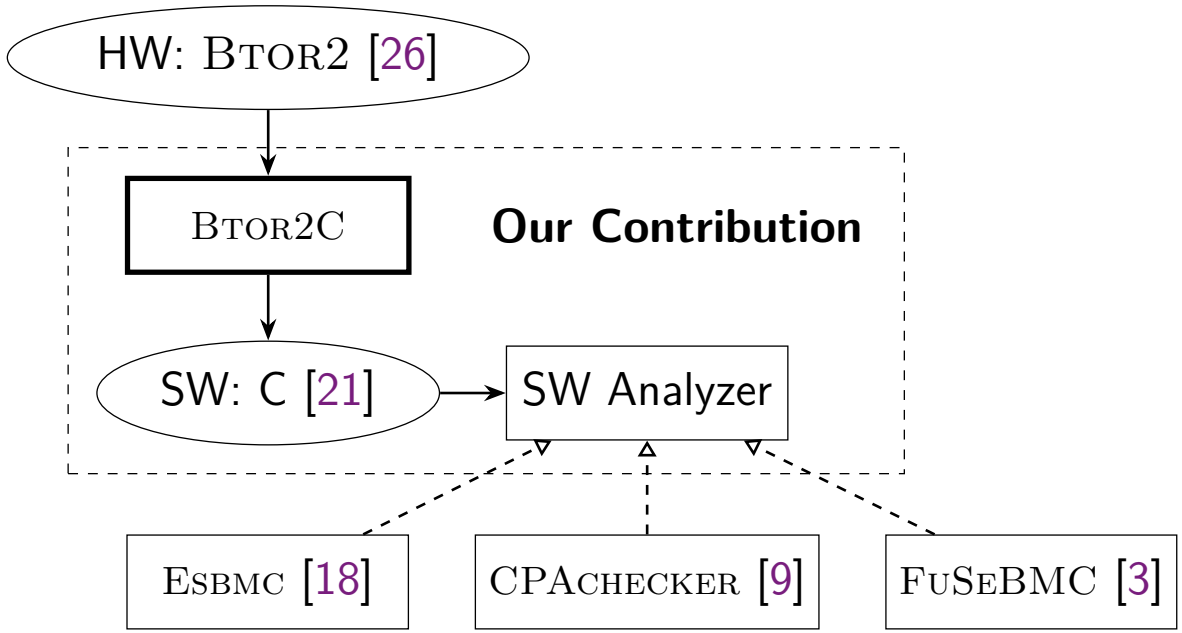
SoSy-Lab Software Systems

About this talk



- **43** HW-verification tasks uniquely solved by SW analyzers in our evaluation

About this talk

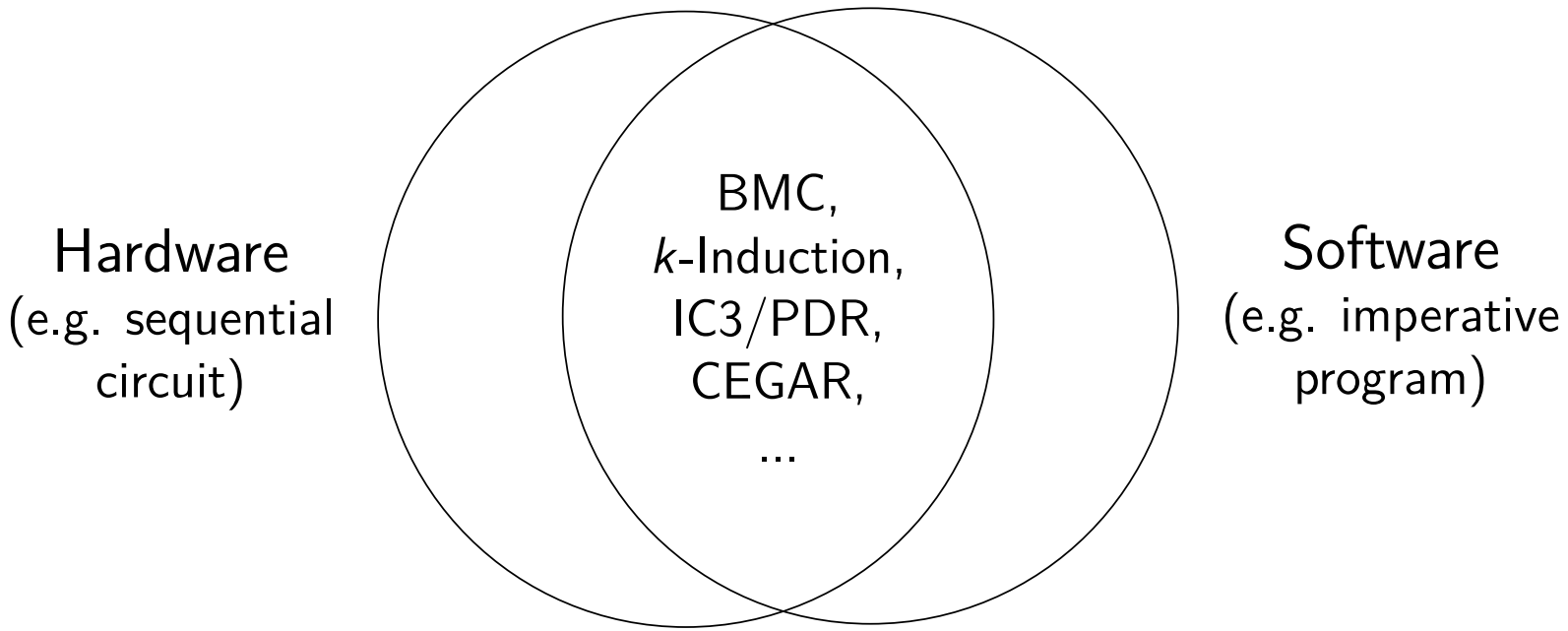


- **43** HW-verification tasks uniquely solved by SW analyzers in our evaluation → enhance HW quality assurance

Outline

Analysis of Computational Systems

- Quality assurance is important for computational systems
- Analysis methods are usually developed for a specific model
- Theoretical foundations and technologies are shared



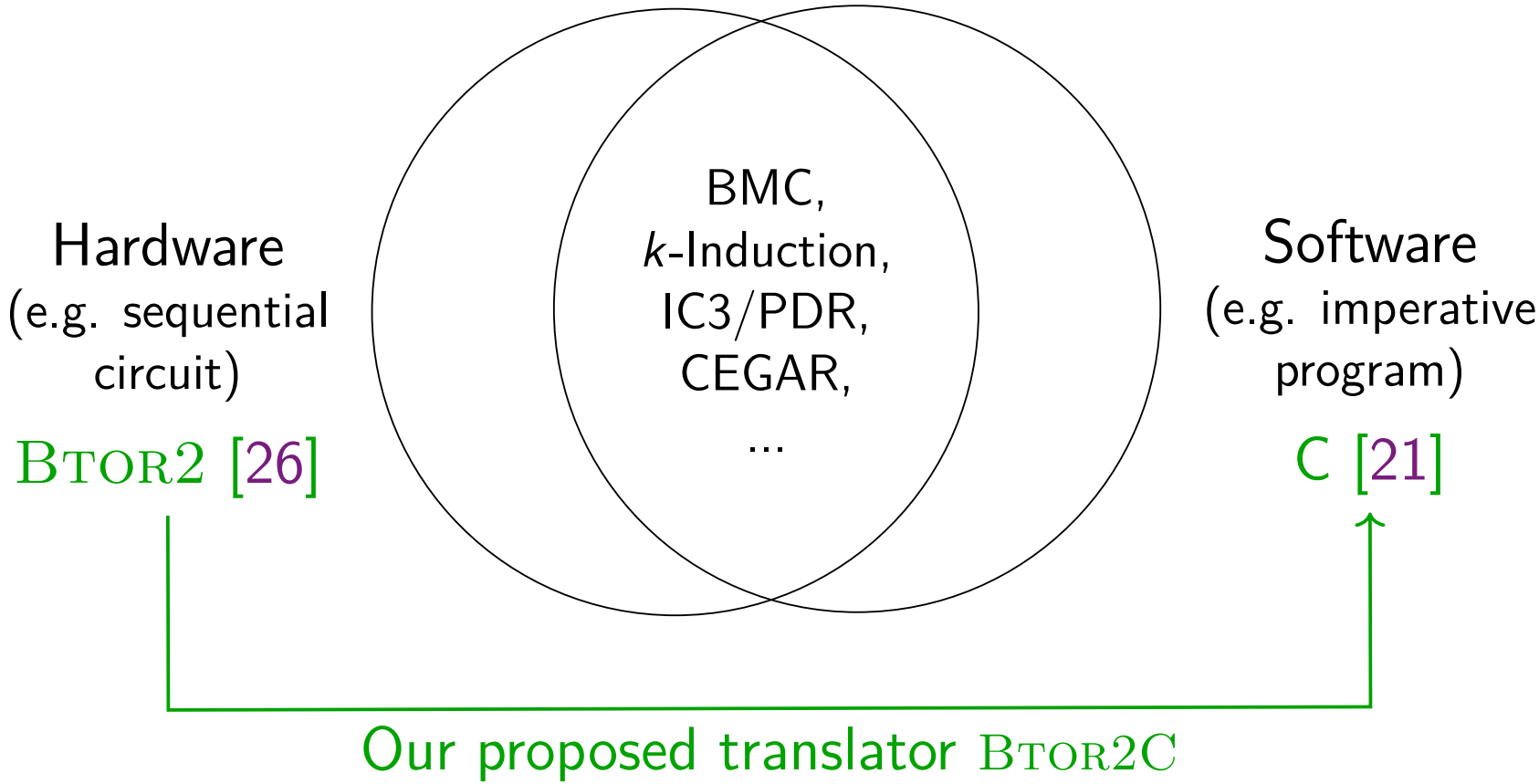
Analysis Algorithms for HW and SW

- HW-verification algorithms adopted for SW
  - BMC [12]
  - *k*-Induction [27]
  - IC3/PDR [13, 8]
- SW-testing algorithms adopted for HW
  - Fuzzing [23]
  - Symbolic execution [22]

Analysis Algorithms for HW and SW

- HW-verification algorithms adopted for SW
  - BMC [12]
  - *k*-Induction [27]
  - IC3/PDR [13, 8]
- SW-testing algorithms adopted for HW
  - Fuzzing [23]
  - Symbolic execution [22]
- Leveraging knowledge from one field to another is **difficult**

Bridging the Gap between HW and SW Analysis



Design Decisions

- Why BTOR2 as frontend?
- Why C as target?
- Why a standalone translator?

Why BTOR2 as Frontend?

- Simple yet expressive
  - A word-level extension to the bit-level format AIGER [10]
  - Bit-precise language for hardware-verification problems
  - Much simpler than Verilog [1]

Why BTOR2 as Frontend?

- Simple yet expressive
  - A word-level extension to the bit-level format AIGER [10]
  - Bit-precise language for hardware-verification problems
  - Much simpler than Verilog [1]
- Extensive tool support
  - Utility tool suite BTOR2TOOLS [25] available
  - YOSYS [28] supports translation from Verilog to BTOR2
  - Used in HWMCC [11] $\Rightarrow$ Supported by many hardware model checkers (and many benchmark tasks available)

Why a standalone translator?

Should we

1. make software analyzers support BTOR2, or
2. build a standalone BTOR2-to-C translator?

Why a standalone translator?

Should we

1. make software analyzers support BTOR2, or
2. build a standalone BTOR2-to-C translator?

We chose 2. because

- Don't have to repeat the effort for each analyzer
- Software analyzers from SV-COMP [5] and Test-Comp [4] are readily available!

Why a standalone translator?

Should we

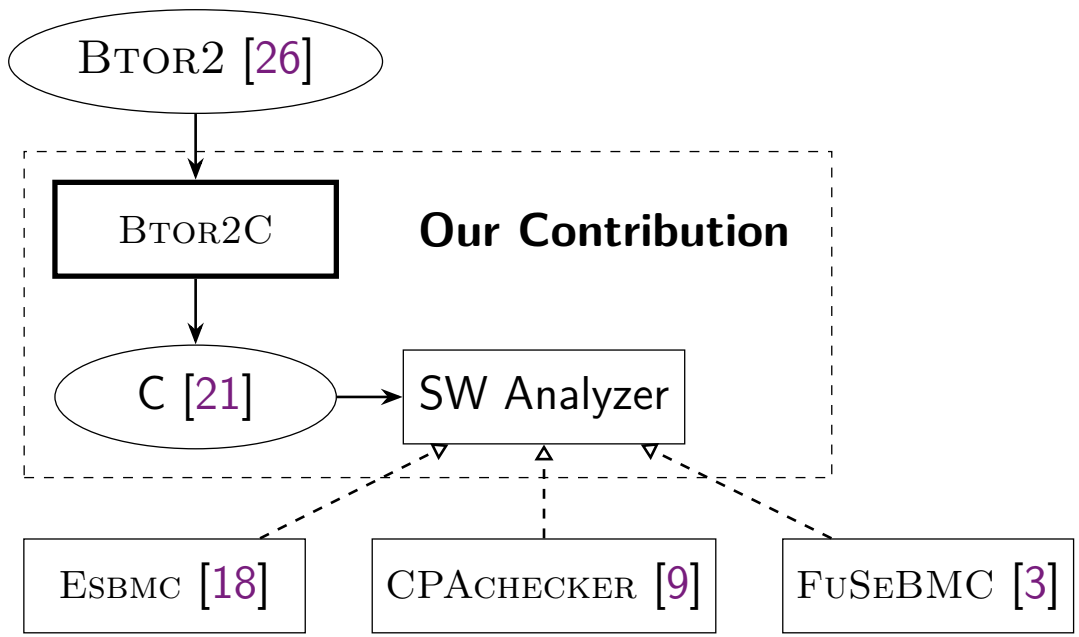
1. make software analyzers support BTOR2, or
2. build a standalone BTOR2-to-C translator?

We chose 2. because

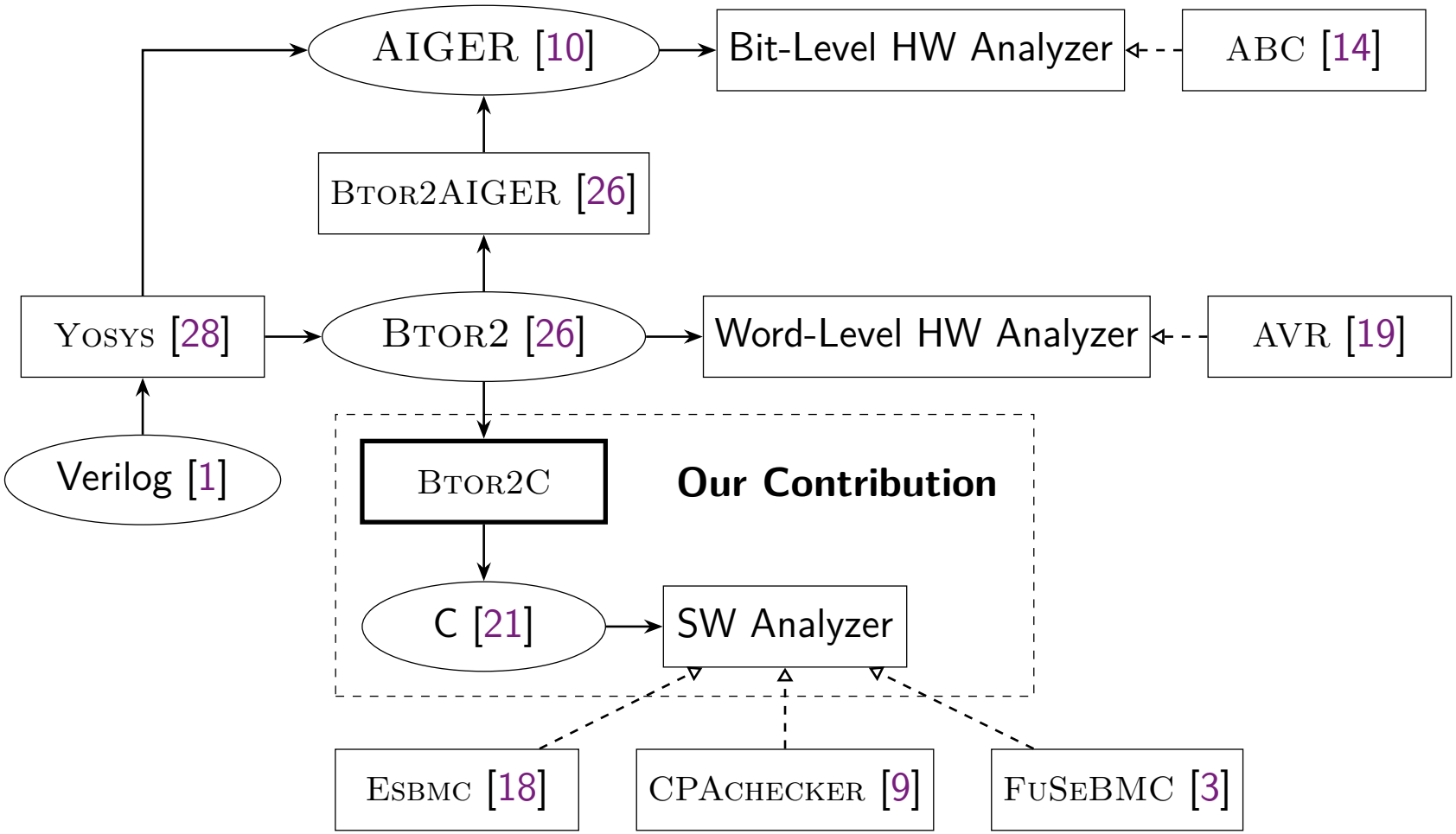
- Don't have to repeat the effort for each analyzer
- Software analyzers from SV-COMP [5] and Test-Comp [4] are readily available!

Why C as target?

Analysis Tool Chain



Analysis Tool Chain



What can Btor2C do for you?

- For HW designers
  - Enhance quality assurance by utilizing SW analyzers

What can Btor2C do for you?

- For HW designers
  - Enhance quality assurance by utilizing SW analyzers
- For developers of SW analyzers
  - Evaluate their algorithms and implementations on HW-verification problems
  - 1224 translated tasks have been contributed to SV-COMP '23 benchmark set

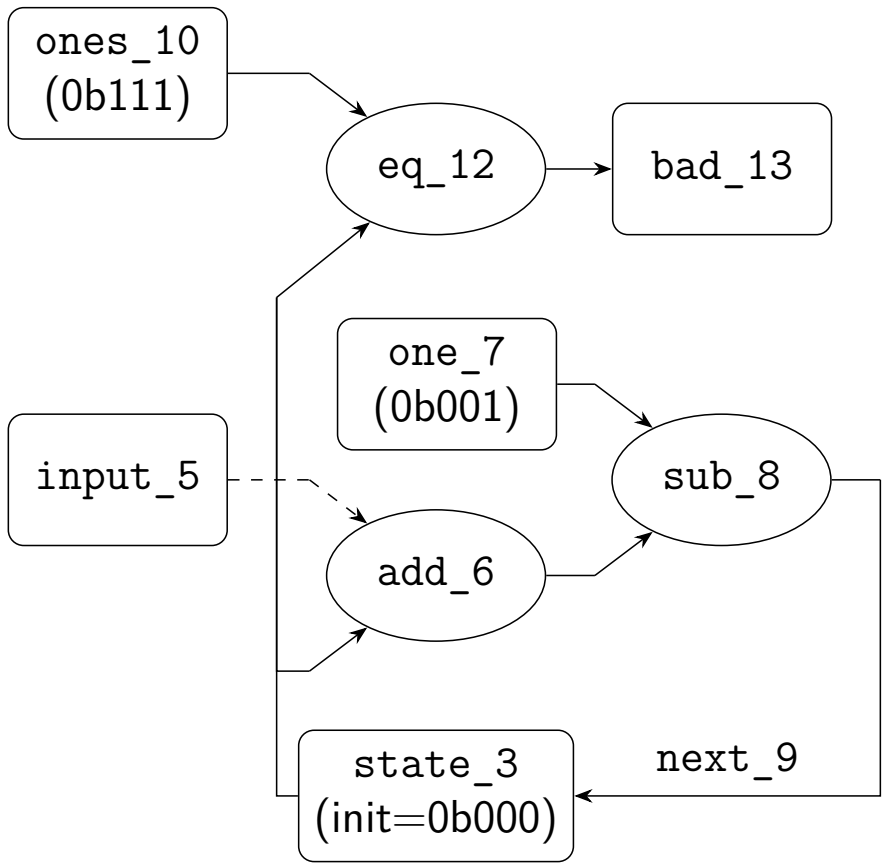
Model-Checking Problem of Reachability Safety

Problem formulation

- For hardware:
  decide whether the safety property holds on all executions of the sequential circuit
- For software:
  decide whether there is an executable program path that reaches the *error* location

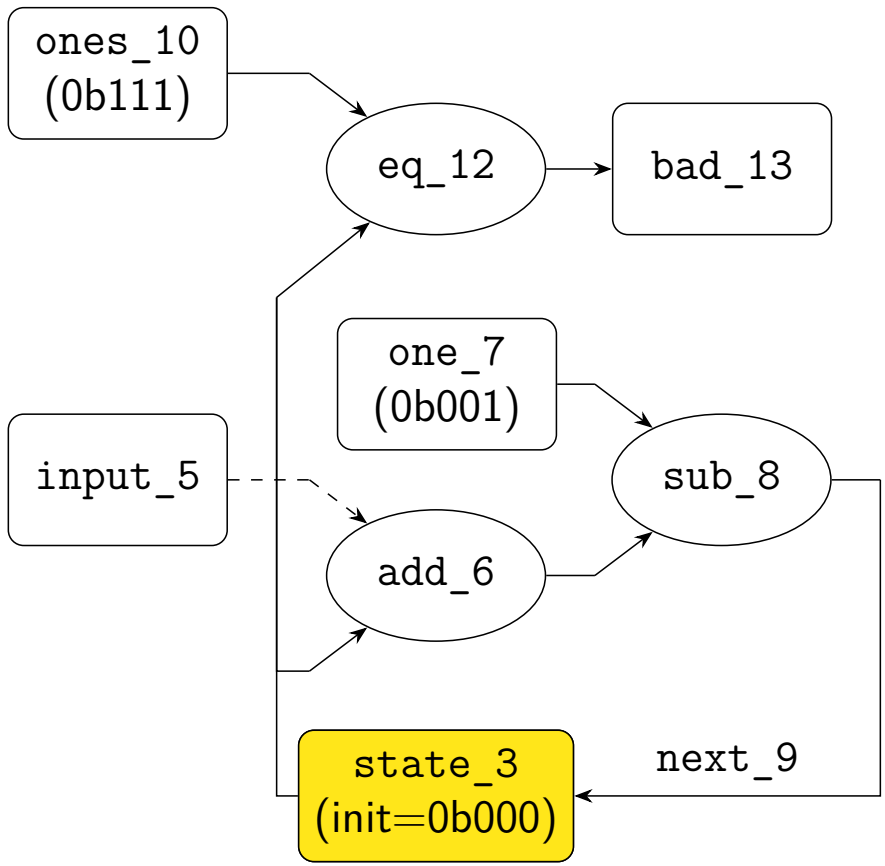
The BTOR2 Language

```
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
```



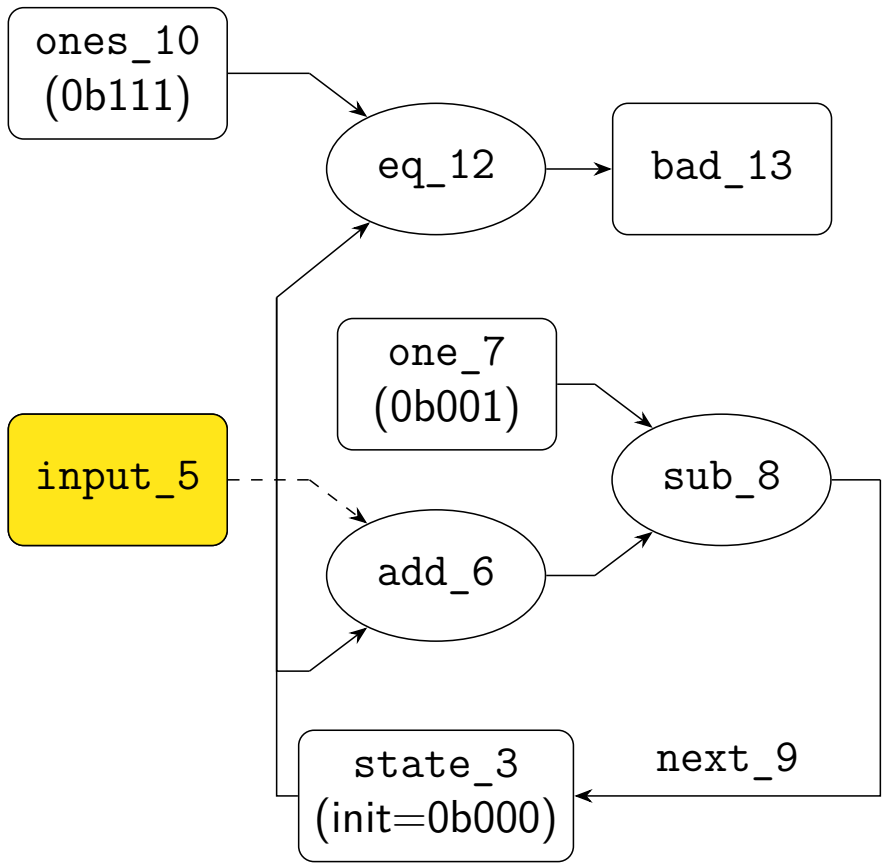
The BTOR2 Language

```
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
```



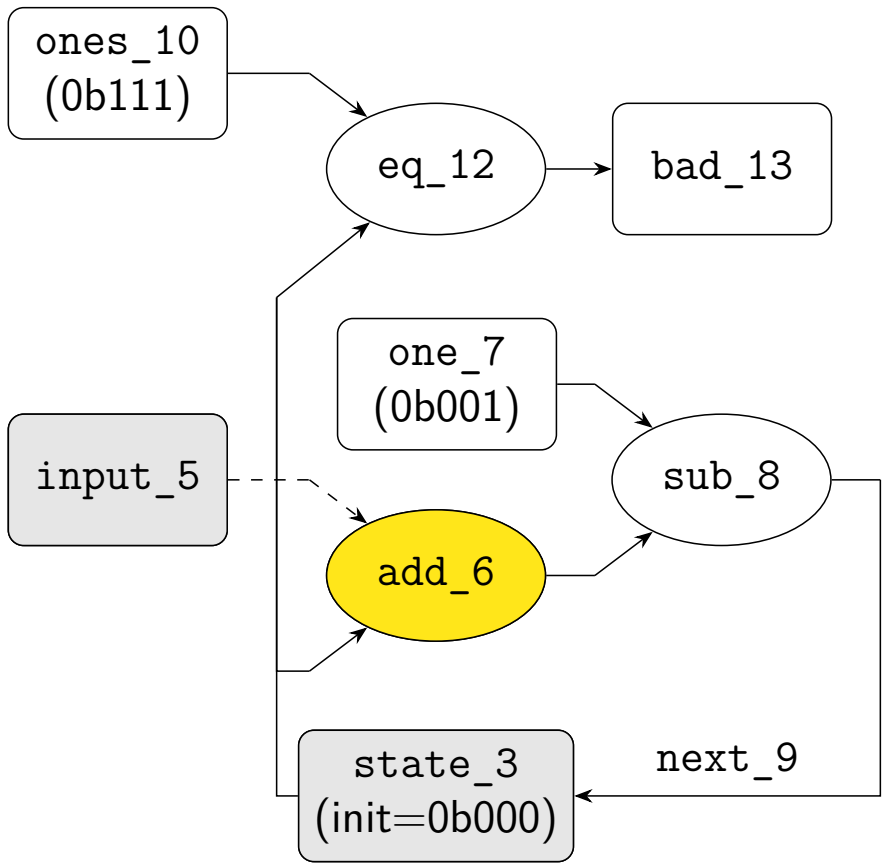
The BTOR2 Language

```
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
```



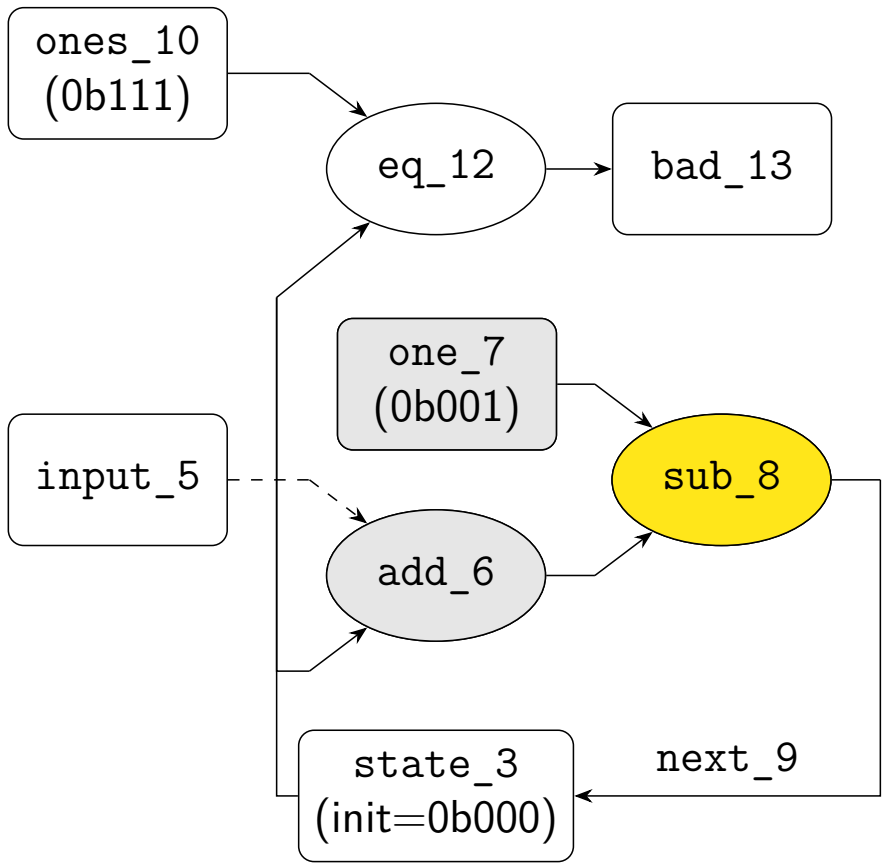
The BTOR2 Language

```
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
```



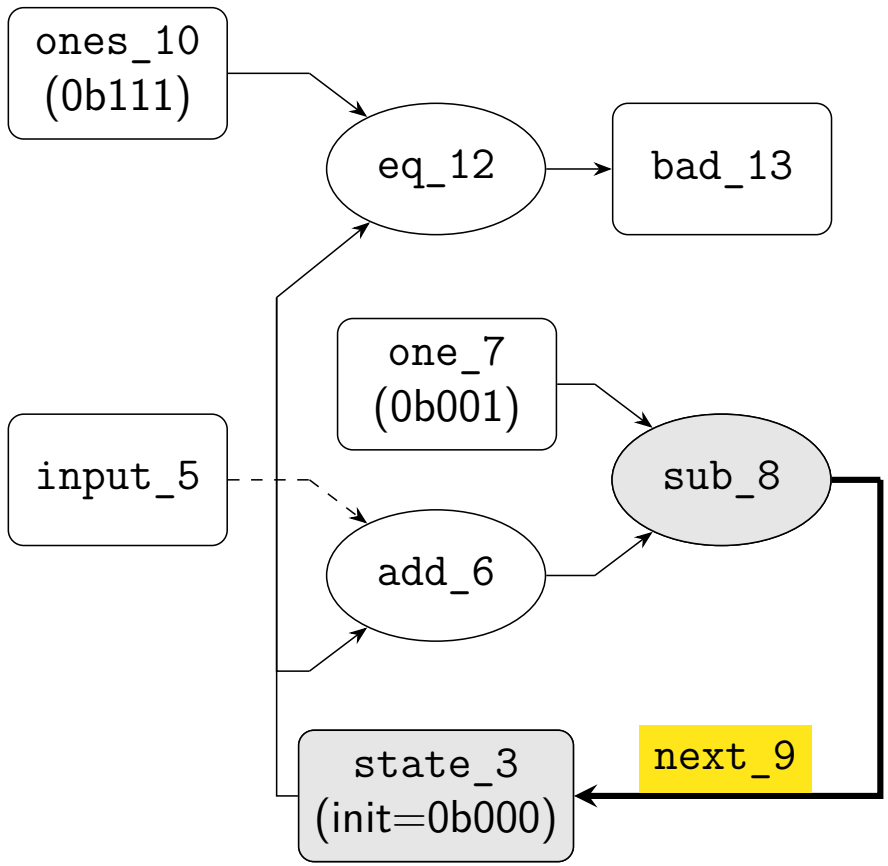
The BTOR2 Language

```
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
```



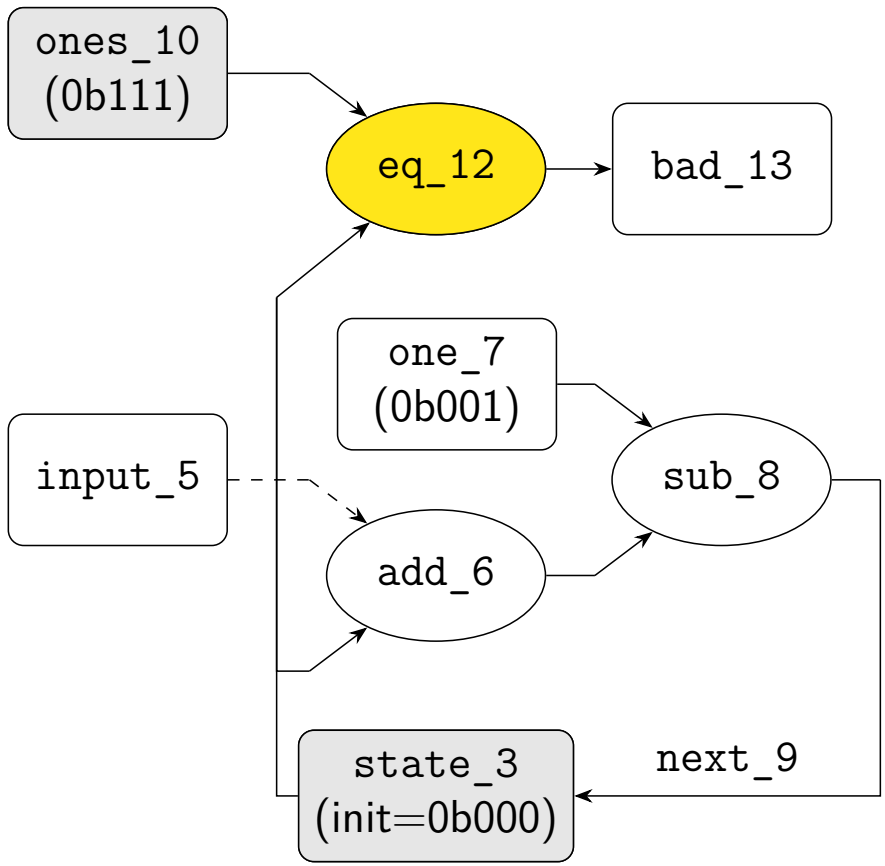
The BTOR2 Language

```
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
```



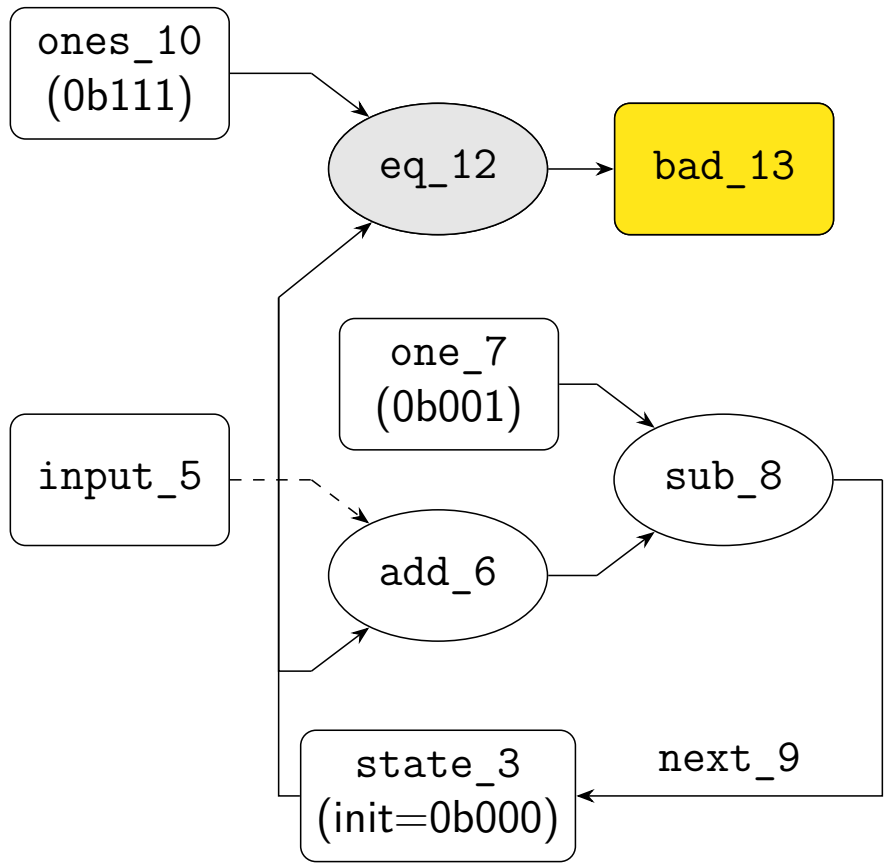
The BTOR2 Language

```
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
```



The BTOR2 Language

```
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
```



Simulating Sequential Circuits with C Programs

A generic program template:

```
1  void main() {
2    // Define sorts and constants
3    // Initialize states
4    for (;;) {
5      // Get external inputs
6      // Evaluate safety property
7      if (bad) {
8        ERROR: abort();
9      }
10     // Compute and assign next states
11   }
12 }
```

Translating BTOR2 to C

```
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
```

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

Translating BTOR2 to C

```
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
```

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

Translating BTOR2 to C

```
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
```

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

Translating BTOR2 to C

```
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
```

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

Translating BTOR2 to C

```
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
```

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

Translating BTOR2 to C

```
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
```

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

Encoding BTOR2 Sorts with C

BTOR2 sorts	C types
`bitvec`	`unsigned {char,short,int,long}` e.g. `sort bitvec 3` → `unsigned char` mask spare bits: `c_var & 0b111`
`array`	static array, e.g. `unsigned int [][]`

Modeling BTOR2 Constructs with C

BTOR2	C
`input`	non-deterministic function
`bad`	`if` guard with an `ERROR` label
`state`	C var in scope `main`
`init`	initialize state var before entering `for`
`next`	update state var at the end of `for`

Implementing BTOR2 Operators in C

A typical line in BTOR2:

```
<nid0> <op> <sid> <nid1> [<nid2 [nid3]>]
```

BTOR2	C
`a add x b c`	`SORT_x var_a = var_b + var_c;`
`a and x b c`	`SORT_x var_a = var_b & var_c;`
`a ite x b c d`	`SORT_x var_a = var_b ? var_c : var_d;`
`a concat x b c`	`SORT_x var_a = var_b << N | var_c;` (`N` is the bit-width of `var_c`)
`a udiv x b c`	`SORT_x var_a = (var_c == 0) ? MAX_VAL : (var_b / var_c);` (`MAX_VAL` is the max value of `SORT_x`)

Bit-masking Spare Bits

- Spare bits if *size*(BTOR2 `bitvec`) $<$ *size*(C `int`)
- To retain precise value: `c_var & BIT_MASK`

Bit-masking Spare Bits

- Spare bits if *size*(BTOR2 `bitvec`) $<$ *size*(C `int`)
- To retain precise value: `c_var & BIT_MASK`
- Bit-masks can be applied
  - **eagerly**: after every operation
  - **lazily**: only when precise values are needed

Bit-masking Spare Bits

- Spare bits if *size*(BTOR2 `bitvec`) $<$ *size*(C `int`)
- To retain precise value: `c_var & BIT_MASK`
- Bit-masks can be applied
  - **eagerly**: after every operation
  - **lazily**: only when precise values are needed
- The latter yields better performance in our experiments

Revisit: Translating BTOR2 to C

```
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
```

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

Revisit: Translating BTOR2 to C

```
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
```

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

Revisit: Translating BTOR2 to C

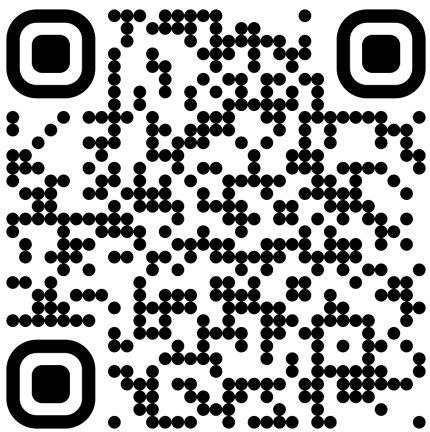
```
1 sort bitvec 3
2 zero 1
3 state 1
4 init 1 3 2
5 input 1
6 add 1 3 5
7 one 1
8 sub 1 6 7
9 next 1 3 8
10 ones 1
11 sort bitvec 1
12 eq 11 3 10
13 bad 12
```

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

Implementation

- A lightweight translator Btor2C
  - Written in C with ∼1600 LOC
  - Use the frontend parser provided by Btor2Tools [25]
  - Compiled binary < 0.25 MB
- Open-source under Apache License 2.0

https://gitlab.com/sosy-lab/software/btor2c



Evaluation

A large-scale experiment was conducted to answer the following research questions

- **RQ1**: Is the BTOR2-to-C translation correct?
  Correctness validated by empirical evaluation

Evaluation

A large-scale experiment was conducted to answer the following research questions

- **RQ1**: Is the BTOR2-to-C translation correct?
  Correctness validated by empirical evaluation
- **RQ2**: How do SW analyzers perform on HWMCC tasks?
  Quite decent! Each analyzer showcases different strength

Evaluation

A large-scale experiment was conducted to answer the following research questions

- **RQ1**: Is the BTOR2-to-C translation correct?
  Correctness validated by empirical evaluation
- **RQ2**: How do SW analyzers perform on HWMCC tasks?
  Quite decent! Each analyzer showcases different strength
- **RQ3**: Can SW analyzers complement HW model checkers?
  *Yes, **43** tasks were uniquely solved by SW verifiers*

State-of-the-art Analyzers

The following tools/algorithms were evaluated

- Winners of HWMCC '20
  - ABC [14] (bit-level): PDR
  - AVR [19] (word-level): PDR
- Top winners of *ReachSafety* category in SV-COMP '22
  - CPAchecker [9]: Predicate Abstraction
  - Esbmc [18]: *k*-Induction
  - VeriAbs [2]: Loop Abstraction + BMC
- Overall winner of Test-Comp '22
  - FuSeBMC [3]: Fuzzing + BMC

Benchmark Set

- Collected from HWMCC '19 and '20
- Total 1498 BTOR2 tasks

	Safe	Unsafe	Total
Bit-vector	868	473	1341
(BV+) Array	140	17	157

- Translation to other formats
  - → AIGER by BTOR2AIGER (array not supported)
  - → C by BTOR2C

Experimental Setup

- OS: Ubuntu 22.04 (64 bit)
- Machine: 3.4 GHz CPU (8 cores) and 33 GB of RAM
- Each task is limited to
  - 2 CPU cores
  - 15 min of CPU time
  - 15 GB of RAM

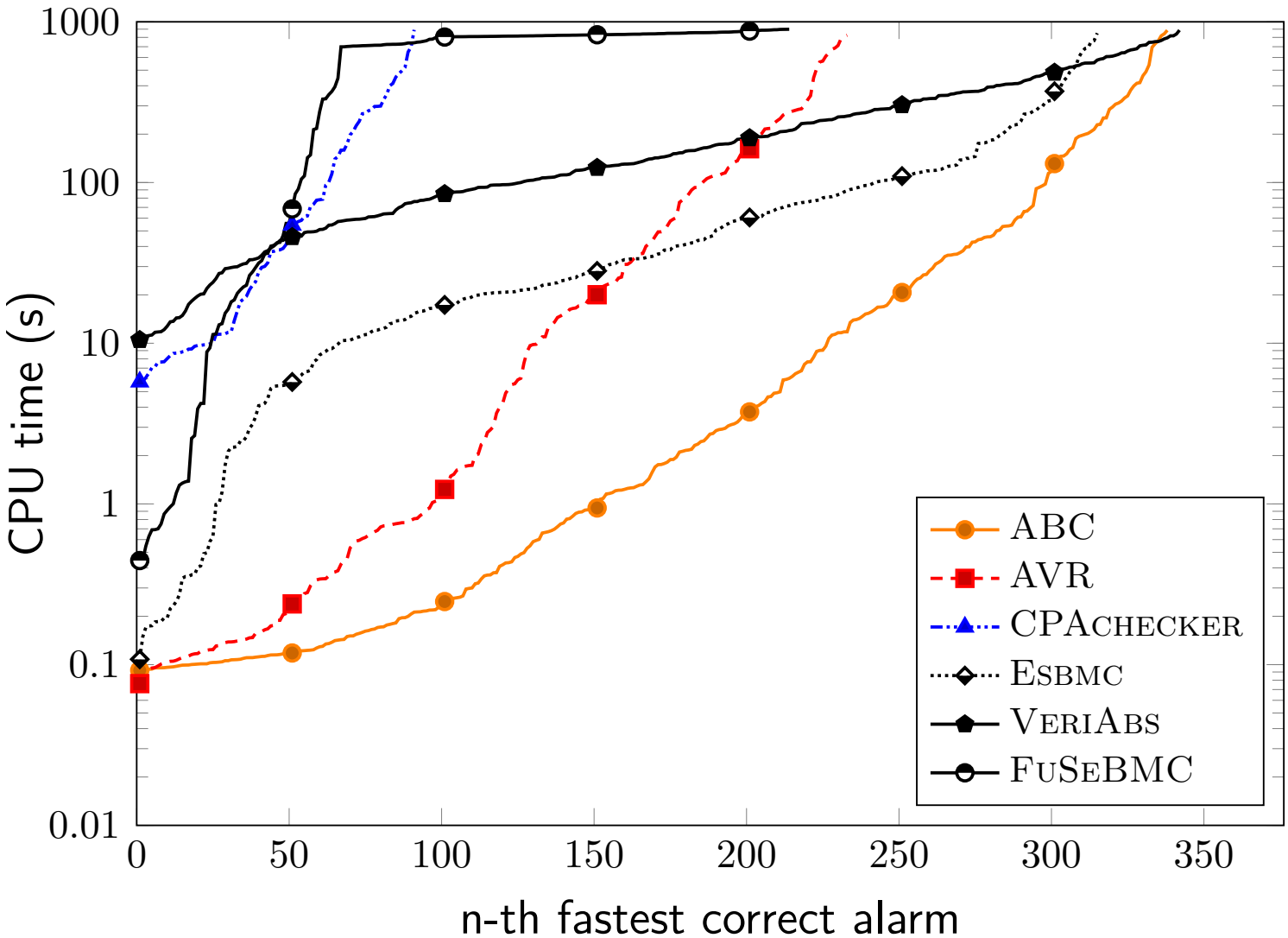
  (reliable resource management by BenchExec[1])

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

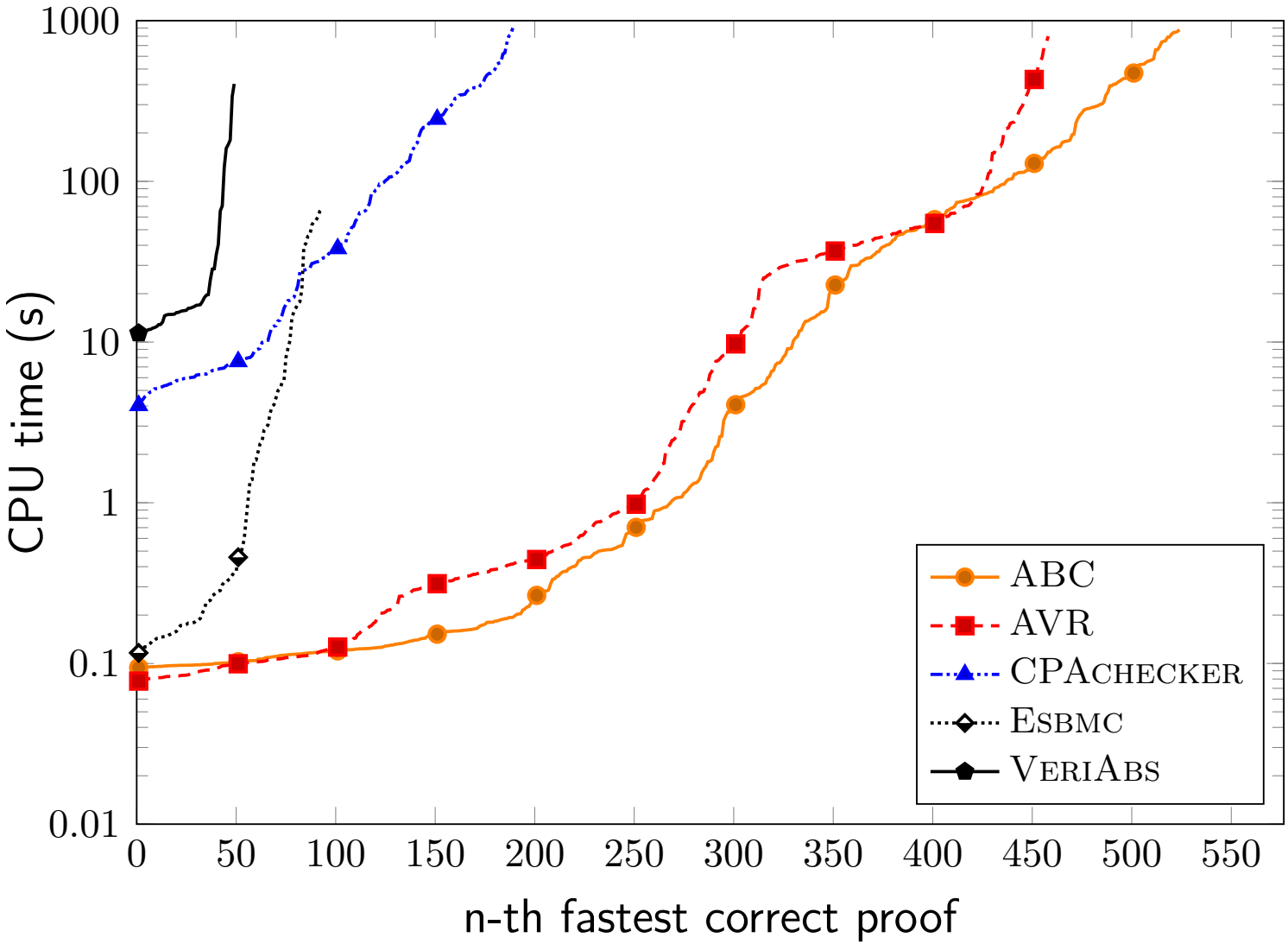
Overall Results

Tool	ABC	AVR	CPAchecker	ESBMC	VeriAbs
Algorithm	PDR	PDR	PA	KI	LA
Input	AIGER	Btor2	C (bit-masking applied lazily)		
Correct results	**862**	736	280	**410**	393
BV proofs	**524**	458	**189**	93	49
BV alarms	**338**	233	91	315	**342**
Array proofs	–	**45**	0	0	0
Array alarms	–	0	0	**2**	**2**
Wrong proofs	0	0	0	0	2
Wrong alarms	0	0	0	0	1
Timeouts	479	559	922	551	1 042
Out of memory	0	3	7	537	4
Other inconclusive	0	200	289	0	56

Quantile Plot: Unsafe Bit-vector Tasks



Quantile Plot: Safe Bit-vector Tasks



Reasons for Performace Difference

- Input: AIGER vs. Btor2 vs. C (with spare bits)
- Representation: transition system vs. control-flow graph
- Backend logical solver
  - ABC: SAT solver
  - Others: SMT solver

Analyzer	Solver
AVR	Yices2 [17]
CPAchecker	MathSAT5 [16]
Esbmc	Boolector3 [26]

Conclusion

- BTOR2C translates BTOR2 circuits to C programs
- With BTOR2C, we can
  - Utilize off-the-shelf software analyzers to check the correctness of hardware designs
  - Improve quality assurance for hardware systems
- In our evaluation, software verifiers were able to solve **43** tasks that hardware verifiers cannot
- The reproduction artifact [6, 7] is available via Zenodo

Future Work

- Improve Btor2C
  - Avoid redundant array copies during `write`
  - Support witness format conversion

Future Work

- Improve Btor2C
  - Avoid redundant array copies during `write`
  - Support witness format conversion
- Bridge the gap from the other direction
  - Translate SW to HW

References I

[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] 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

[5] 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

[6] 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

References II

[7] 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

[8] Beyer, D., Dangl, M.: Software verification with PDR: An implementation of the state of the art. In: Proc. TACAS (1). pp. 3–21. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_1

[9] 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

[10] 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

[11] Biere, A., Froleyks, N., Preiner, M.: 11th Hardware Model Checking Competition (HWMCC 2020). http://fmv.jku.at/hwmcc20/, accessed: 2023-01-29

[12] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2

References III

[13] Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI. pp. 70–87. LNCS 6538, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

References IV

[19] 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

[20] Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: VERILOG2SMV: A tool for word-level verification. In: Proc. DATE. pp. 1156–1159 (2016), https://ieeexplore.ieee.org/document/7459485

[21] 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

[22] King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252

[23] Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Trans. Software Eng. **47**(11), 2312–2331 (2021). https://doi.org/10.1109/TSE.2019.2946563

[24] 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

References V

[25] 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

[26] 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

[27] Wahl, T.: The k-induction principle (2013), http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf

[28] Wolf, C.: Yosys open synthesis suite. https://yosyshq.net/yosys/, accessed: 2023-01-29

[29] 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

Related Work

- Translating hardware to software
  - V2C [24] translates Verilog to C
  - BTOR2LLVM[1] and BTOR2CHC[2] translate BTOR2 to LLVM-IR and Constrained Horn Clauses, respectively
- Compiling hardware to intermediate representation
  - YOSYS [28] translates Verilog to AIGER and BTOR2
  - QUTERTL [29] translates Verilog/VHDL to BTOR
  - VERILOG2SMV [20] compiles Verilog into SMV [15]

[1] https://github.com/stepwise-alan/btor2llvm
[2] https://github.com/stepwise-alan/btor2chc

The BTOR2 Language

```
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
```

- Each line starts with a unique index (variable id)
- `sort`: `bitvec` or `array`
- `input`: external input to the circuit
- Registers and memories:
  - `state`: declare a state variable
  - `init`/`next`: initialize/update
- `bad`: bad state property
- Constants: `zero`, `one(s)`, `const`
- Operators: `add`, `sub`, `eq`, ...

Limitations

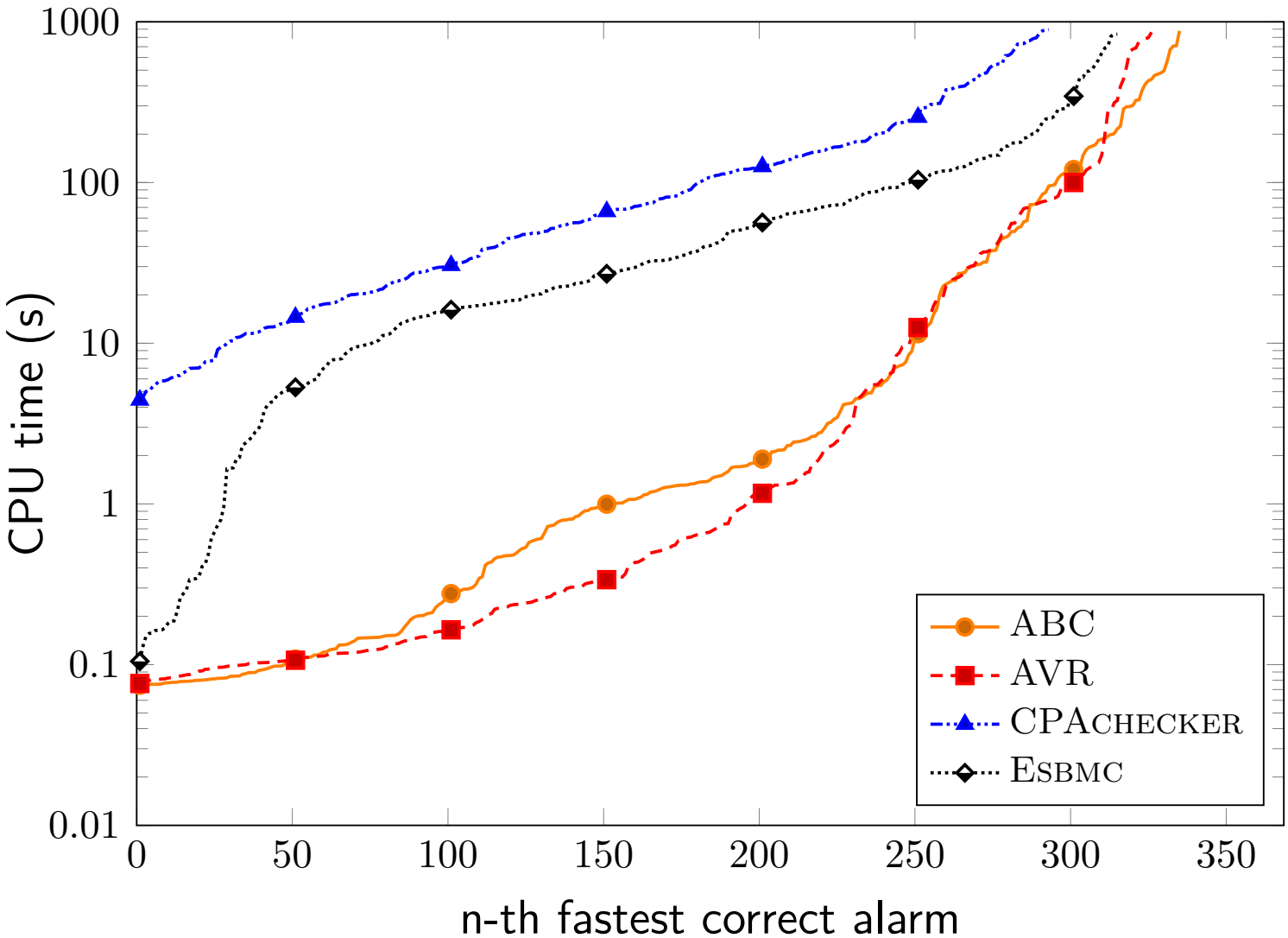
Unsupported BTOR2 features

- Bit-vector sort of size larger than 64 bits
- Fairness constrains: `fair`
- Liveness properties: `justice`
- Overflow detection: `{add,div,mul,sub}o`

Benchmark Sources

- Microprocessor
  - picoJava
  - RIDECORE (RISC-V)
- Application
  - Cyclic redundancy check
  - Huffman coding
  - mutual exclusion
- Industry

Quantile Plot: Comparing BMC Implementations



ESBMC-KI vs. AVR-KI