# Bridging Hardware and Software Verification Witnesses

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

LMU Munich, Germany

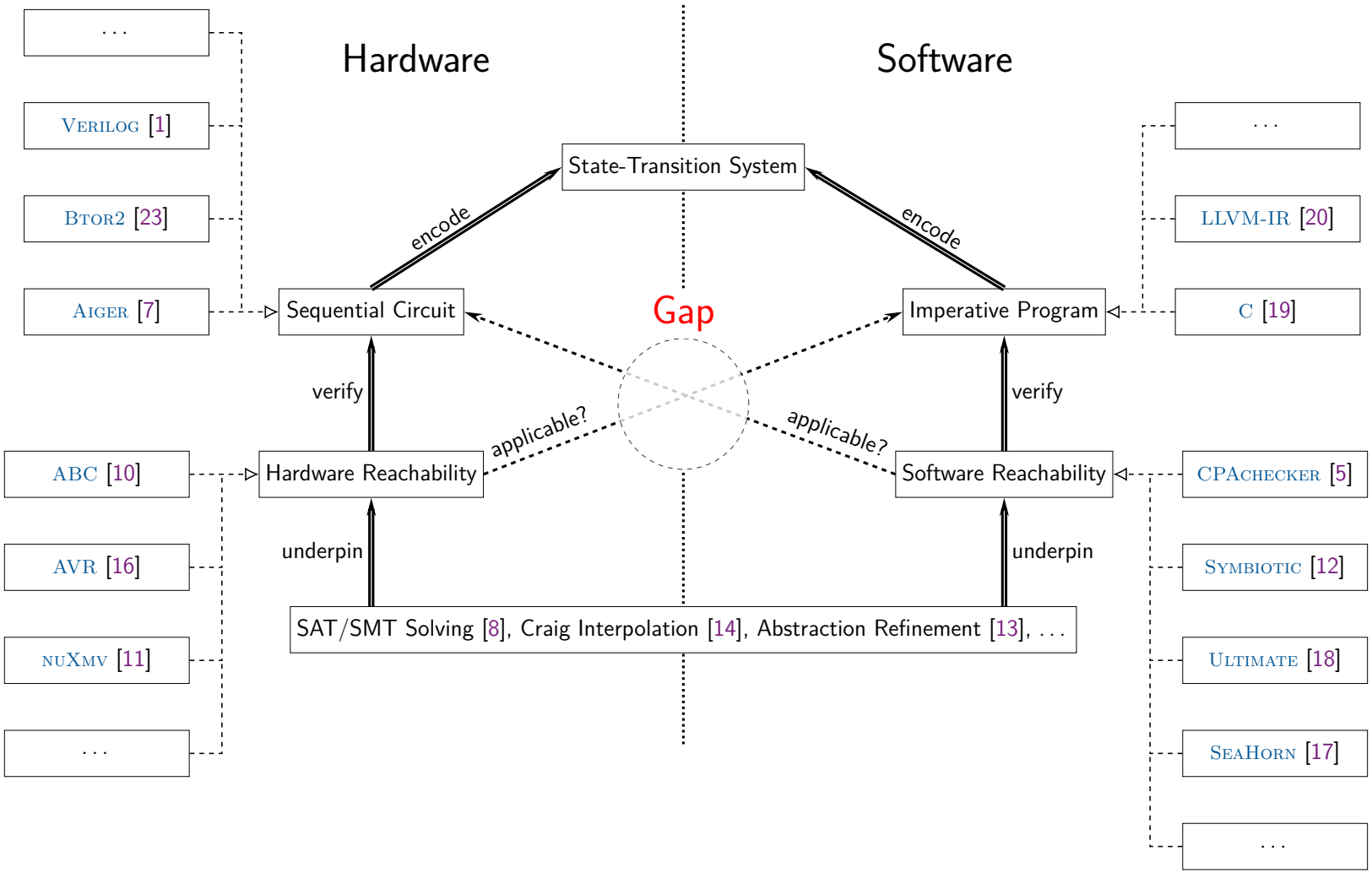
VeWit 2023-07-17



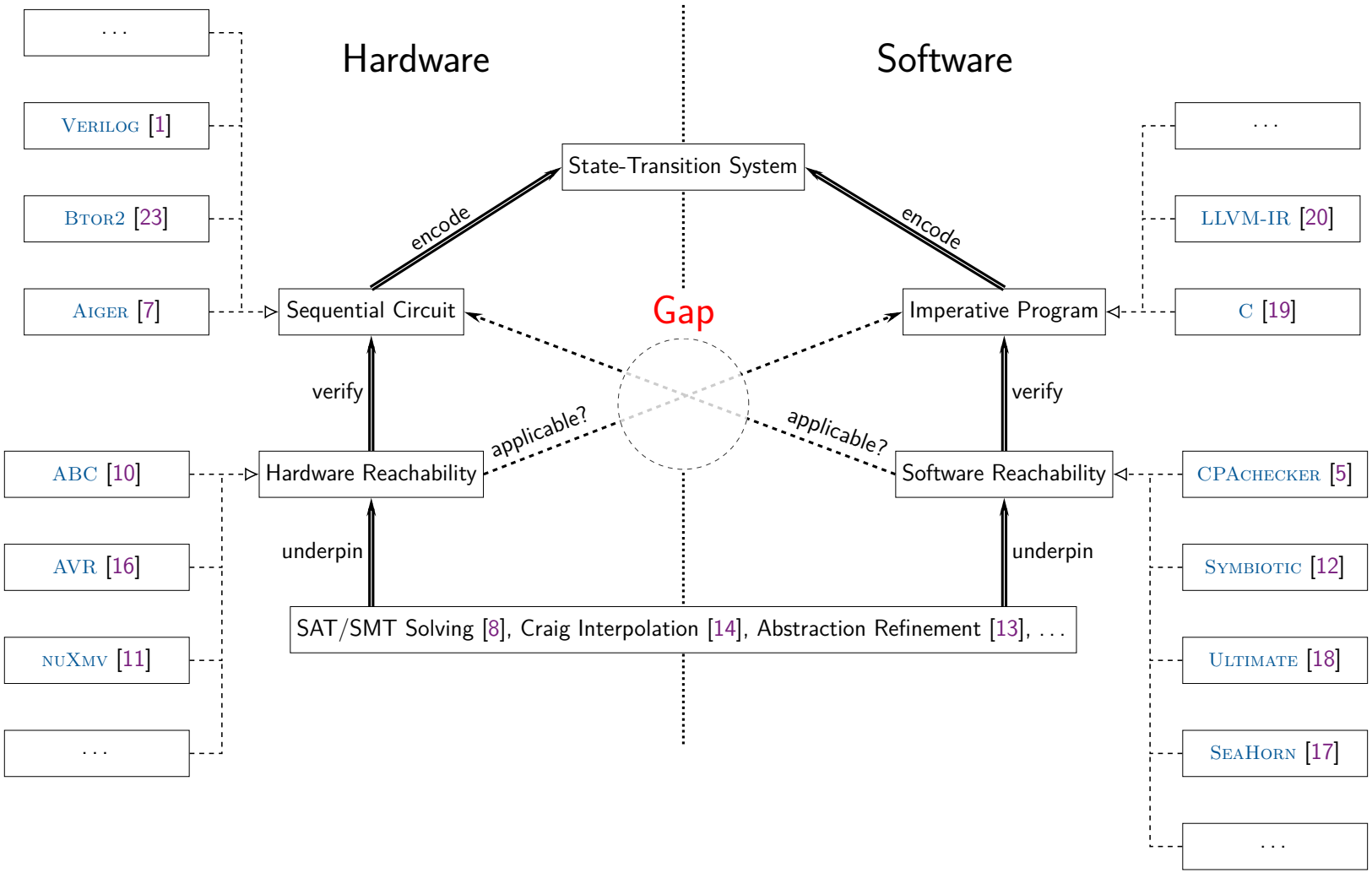
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

SoSy-Lab Software Systems

# Closing the Gap between HW and SW Analysis



# Closing the Gap between HW and SW Analysis



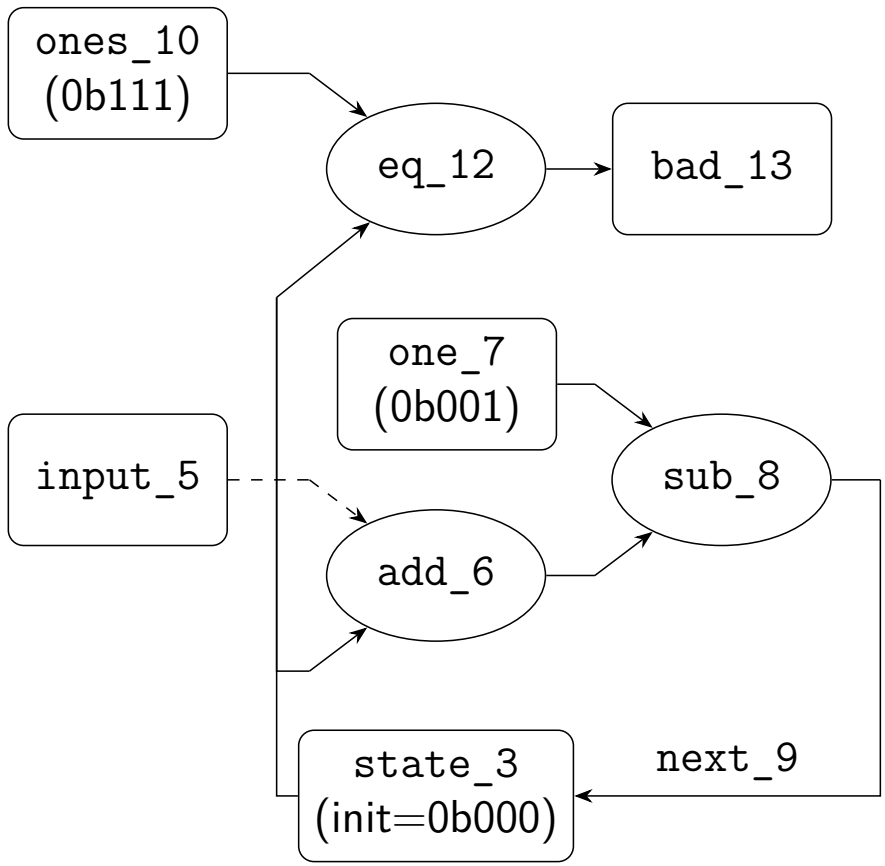

- Information exchange and explainability via **Witnesses**

# Outline

1. The Btor2 Language and Btor2C
2. Translating Software Witnesses to Btor2
3. Discussion

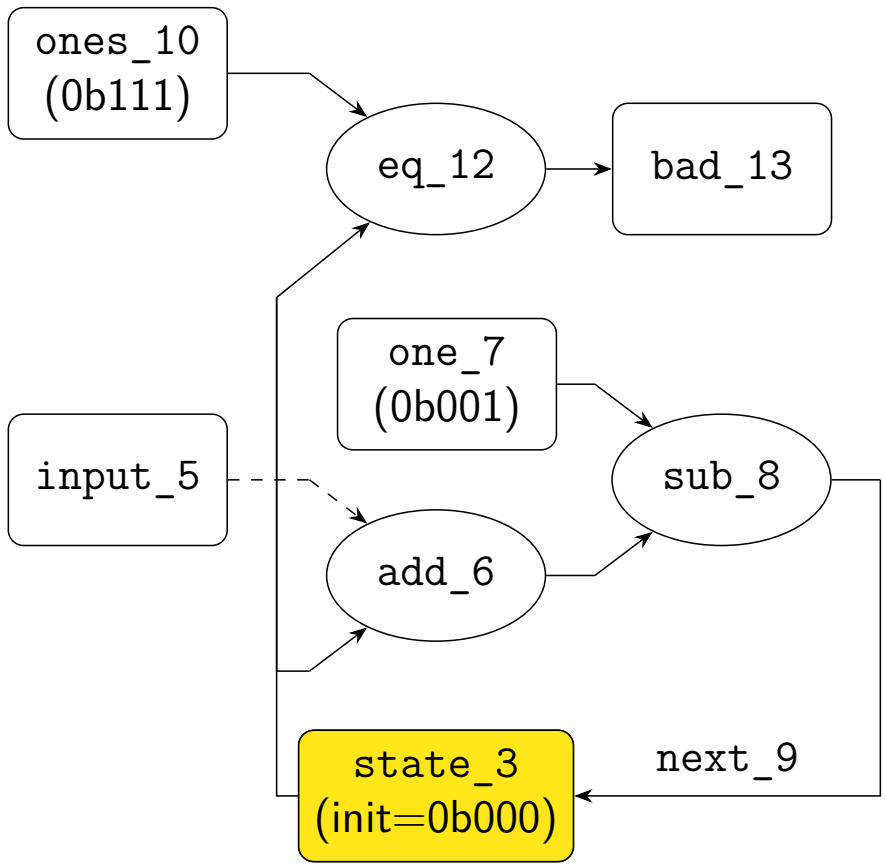
# The BTOR2 Language

```
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
```



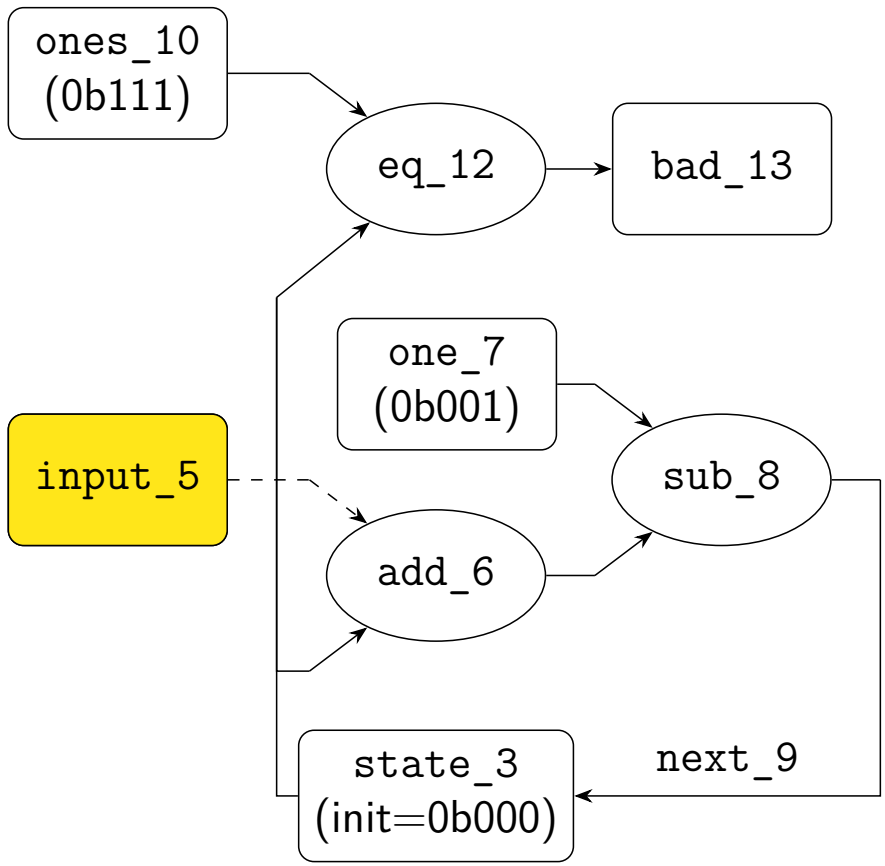
# The BTOR2 Language

```
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
```



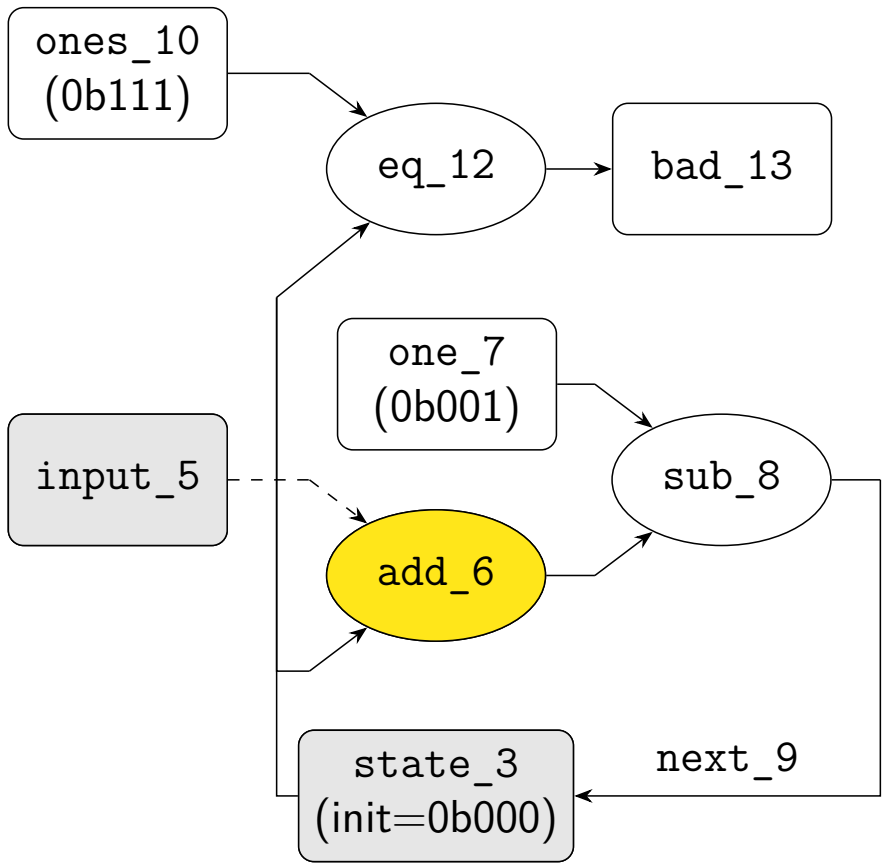
# The BTOR2 Language

```
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
```



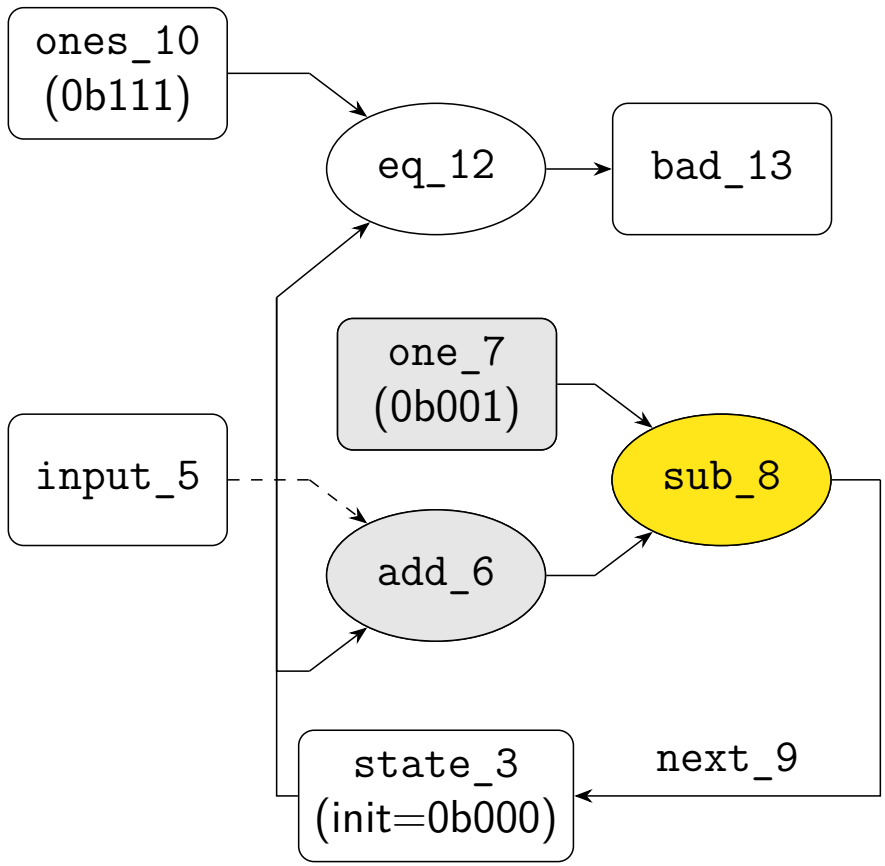
# The BTOR2 Language

```
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
```



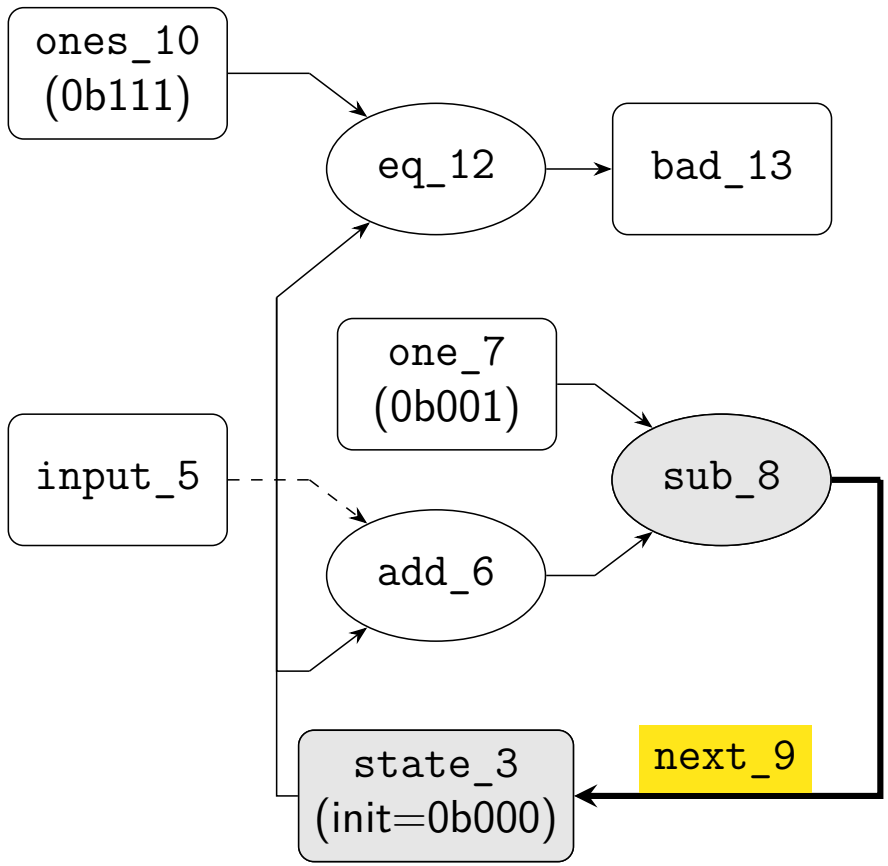
# The BTOR2 Language

```
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
```



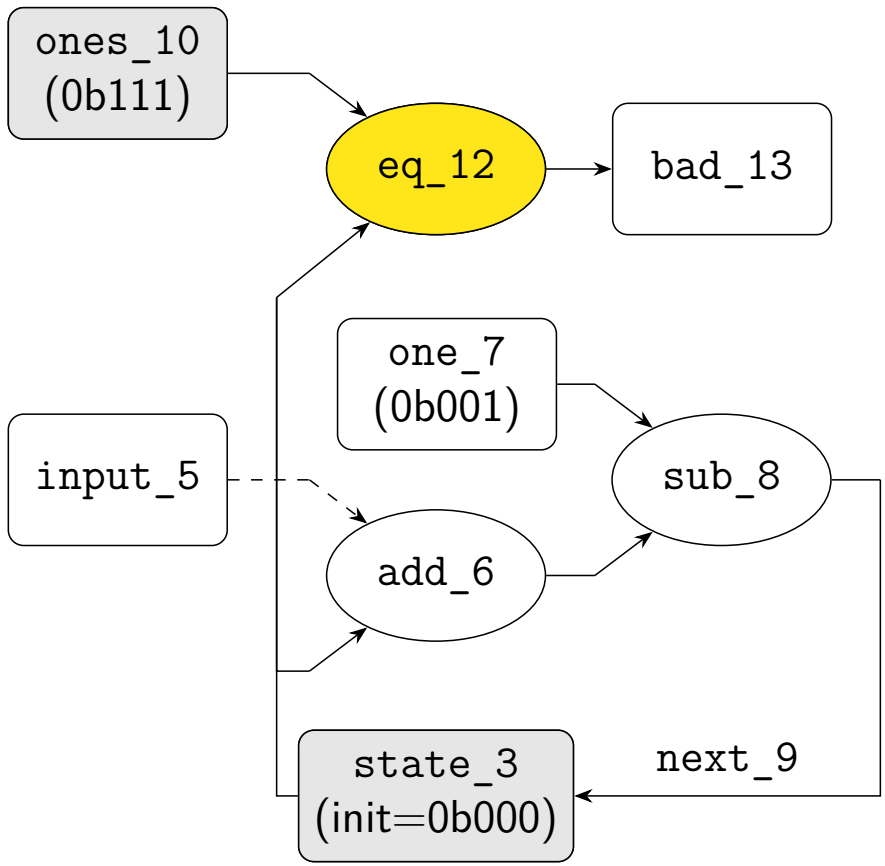
# The BTOR2 Language

```
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
```



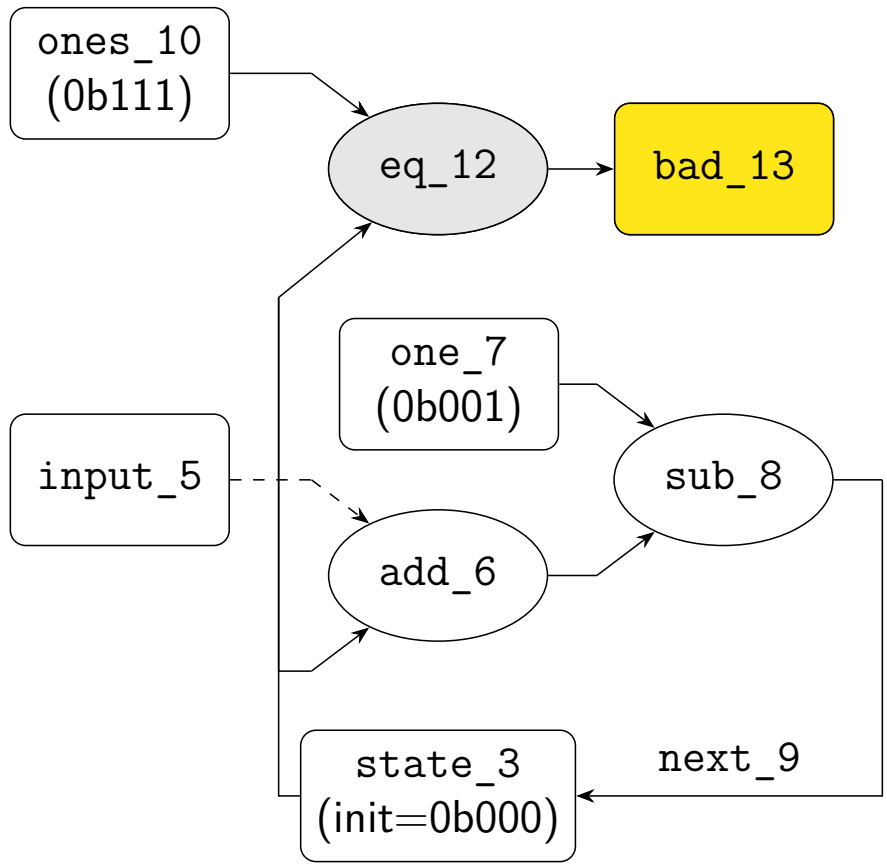
# The BTOR2 Language

```
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
```



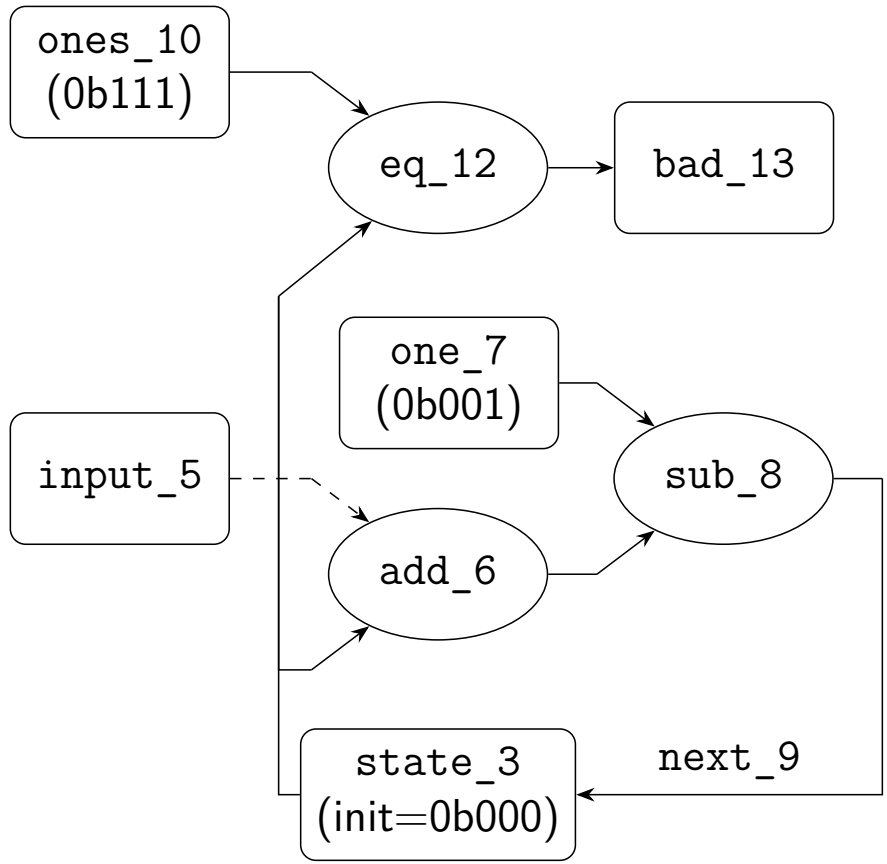
# The BTOR2 Language

```
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
```



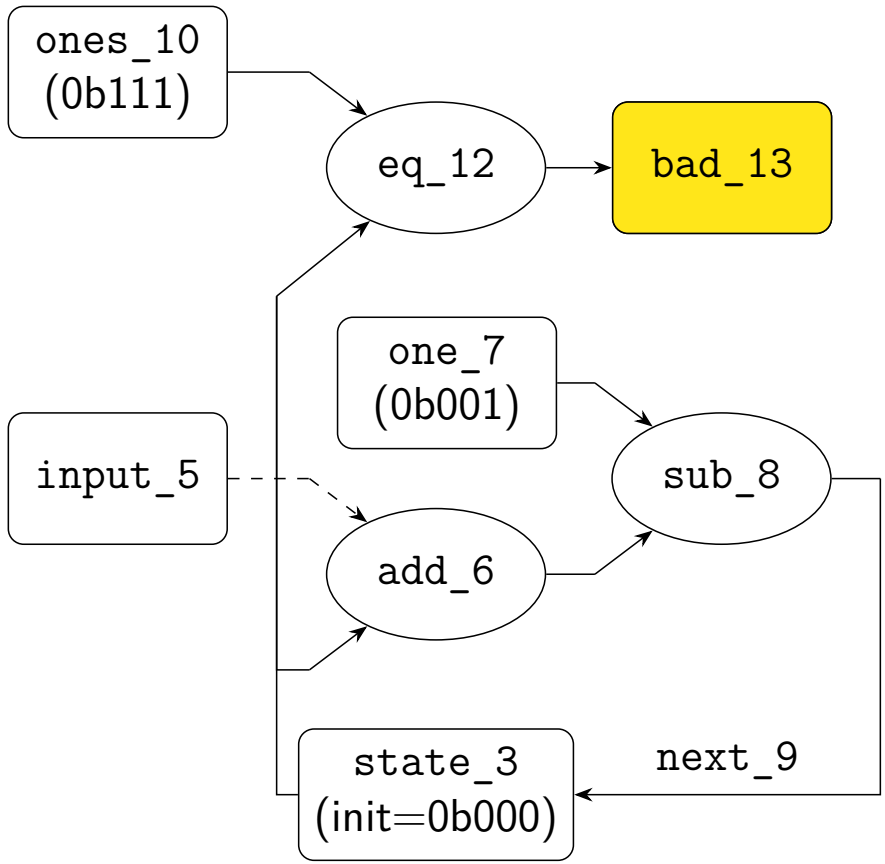
# Btor2 Violation Witness

```
sat
b0
@0
0 100 input_5@0
@1
0 101 input_5@1
@2
.
```



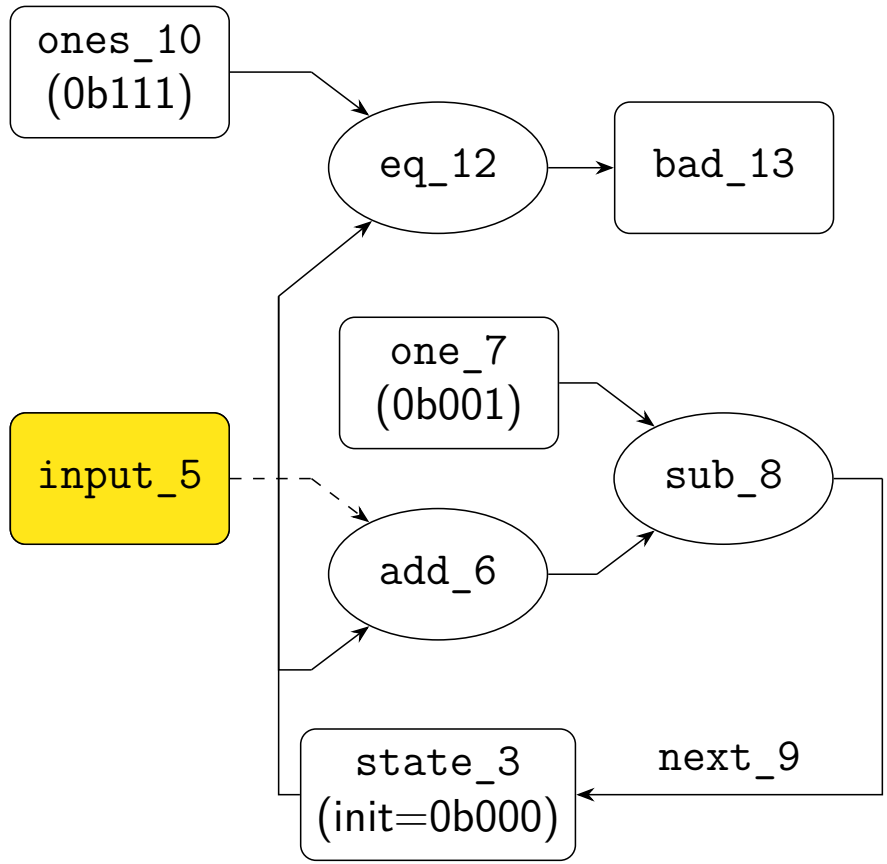
# BTOR2 Violation Witness

```
sat
b0
@0
0 100 input_5@0
@1
0 101 input_5@1
@2
.
```



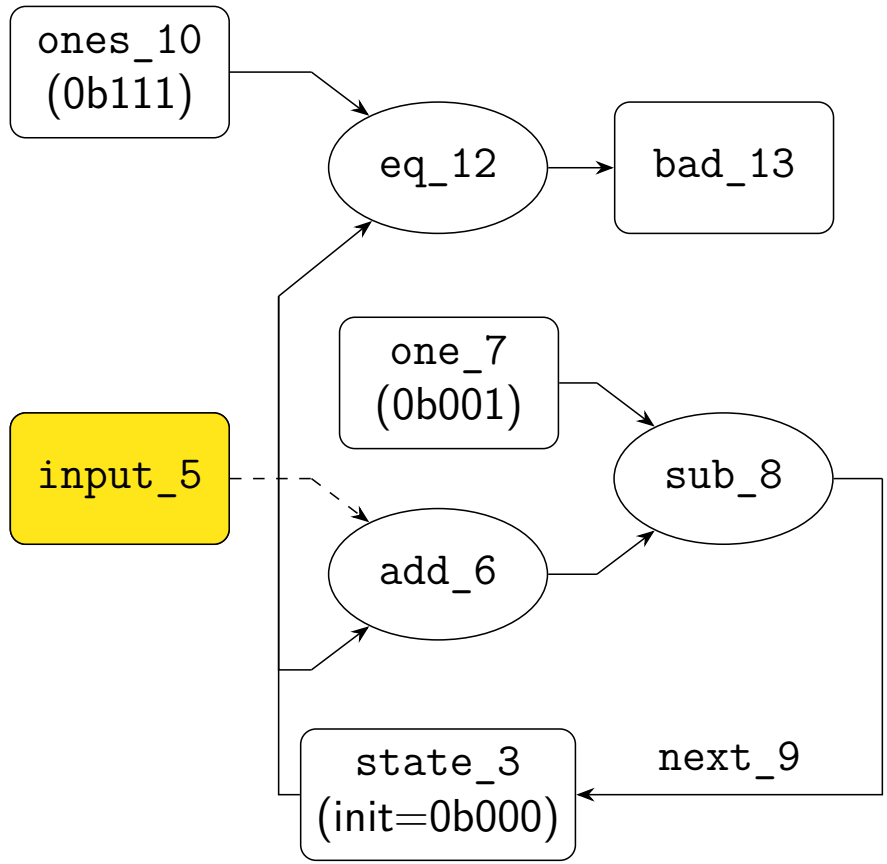
# Btor2 Violation Witness

```
sat
b0
@0
0 100 input_5@0
@1
0 101 input_5@1
@2
.
```



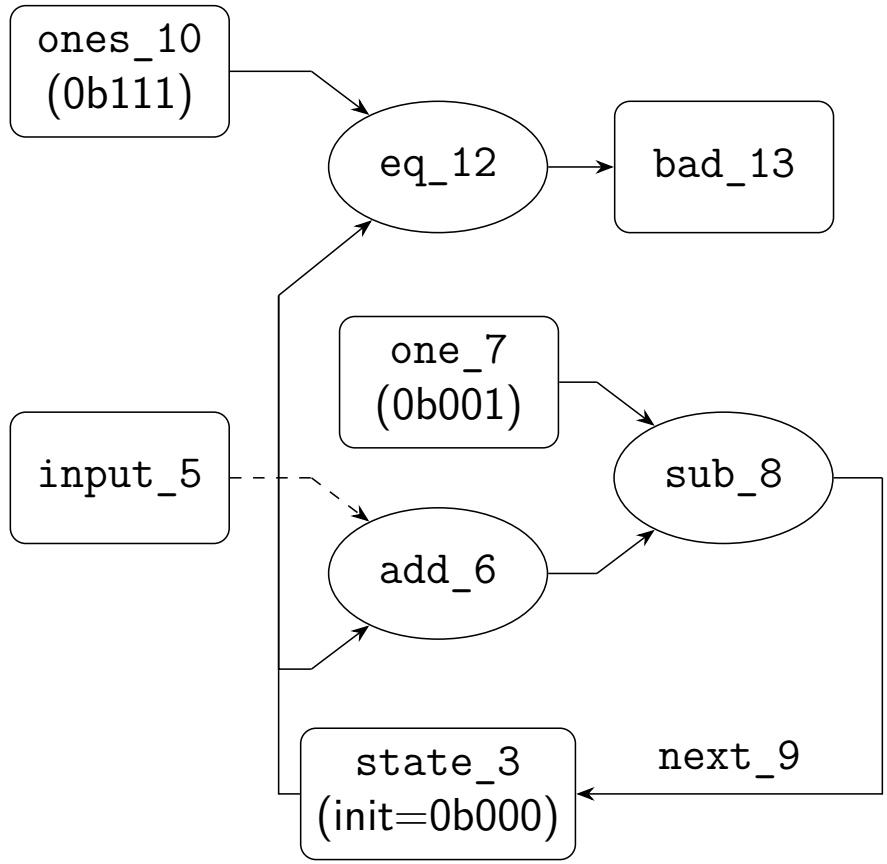
# BTOR2 Violation Witness

```
sat
b0
@0
0 100 input_5@0
@1
0 101 input_5@1
@2
.
```



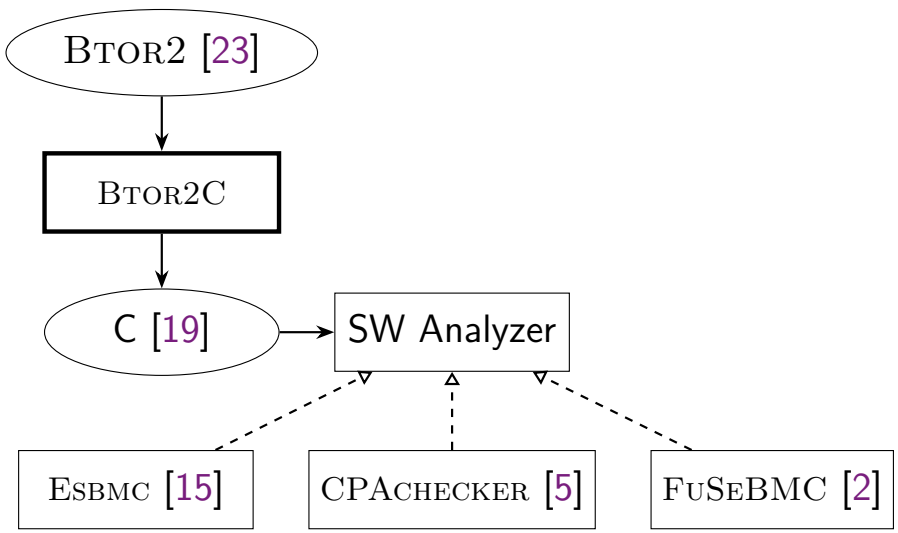
# BTOR2 Violation Witness

```
sat
b0
@0
0 100 input_5@0
@1
0 101 input_5@1
@2
.
```



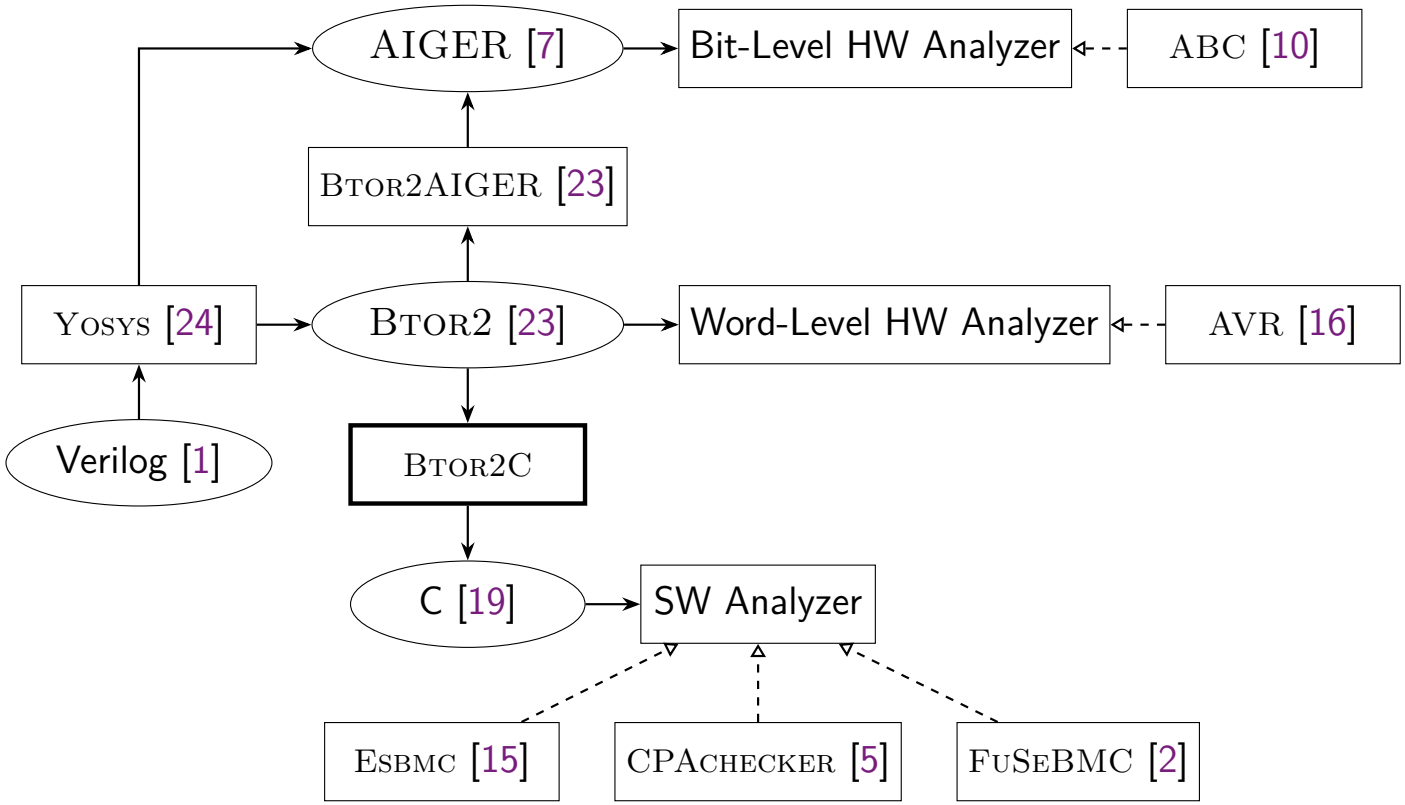

- A violation witness can be simulated by BTORSIM
- No format for **correctness witnesses** in BTOR2 [23]

# Btor2C: Translator from Btor2 to C [3]




[1] Available in `sv-benchmarks`: `hardware-verification-{bv,array}`

# Btor2C: Translator from Btor2 to C [3]




- Btor2 circuits translated to C programs[1]
- HW/SW tools compared on same tasks (and translations)

[1] Available in `sv-benchmarks`: `hardware-verification-{bv,array}`

# 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 }
```

# Motivation

# Motivation

- Correctness guarantee of Btor2C?
  - Btor2 circuit vs. Translated C program

# Motivation

- Correctness guarantee of Btor2C?
  - Btor2 circuit vs. Translated C program
- Information exchange from SW tools to HW designers?

# Motivation

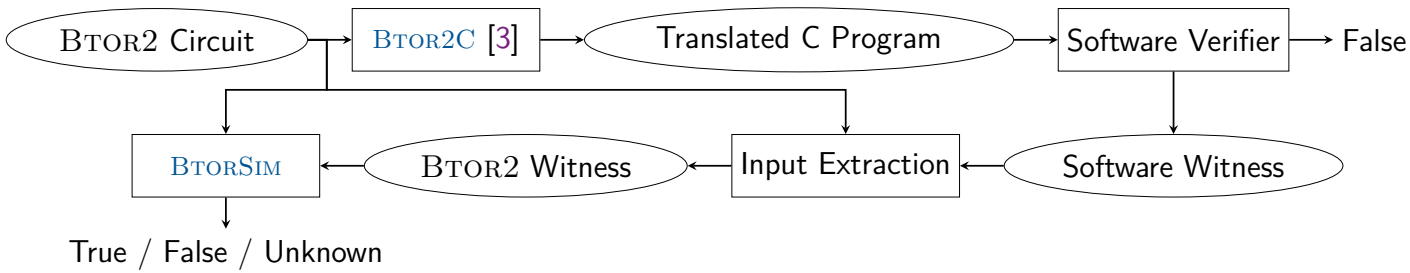
- Correctness guarantee of Btor2C?
  - Btor2 circuit vs. Translated C program
- Information exchange from SW tools to HW designers?
- Validating software witnesses for translated C programs?

# Motivation

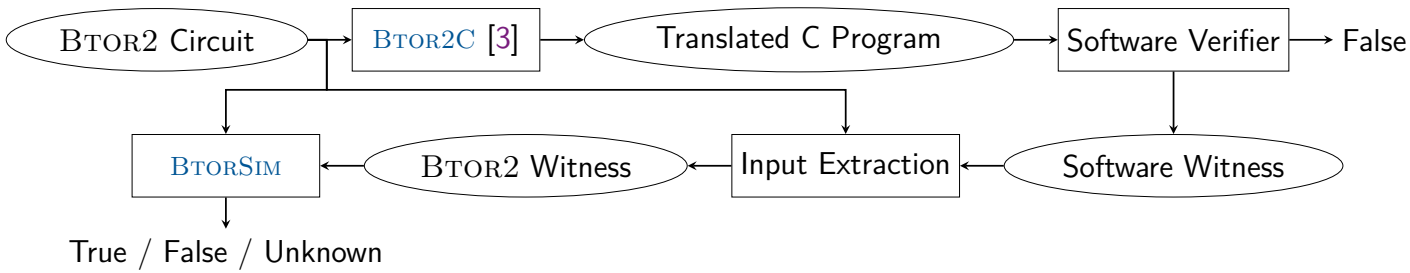
- Correctness guarantee of Btor2C?
  - Btor2 circuit vs. Translated C program
- Information exchange from SW tools to HW designers?
- Validating software witnesses for translated C programs?

Translating software witnesses to Btor2 witnesses!

# Violation Witnesses

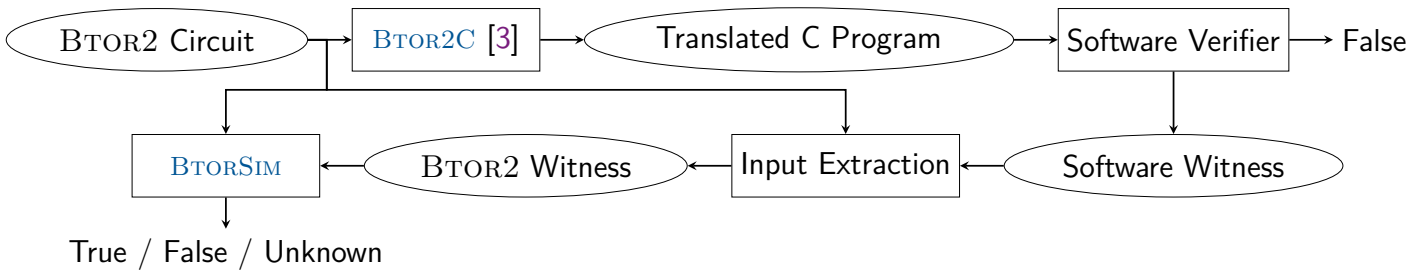


# Violation Witnesses



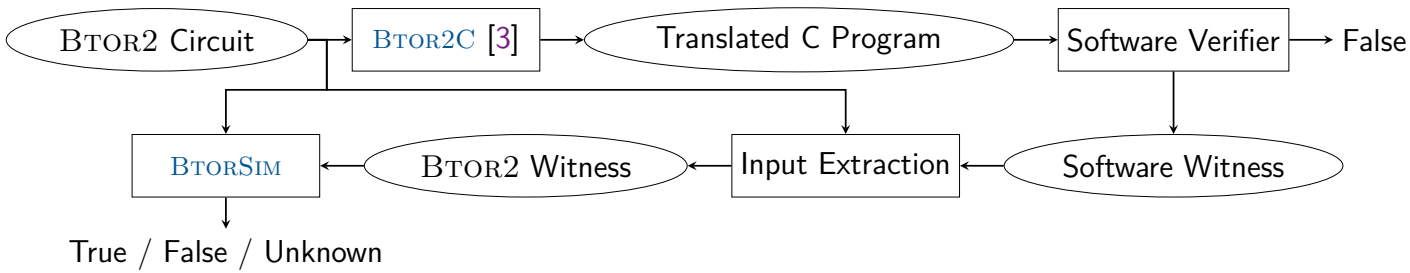

- Extract input assignments (and state initializations)

# Violation Witnesses



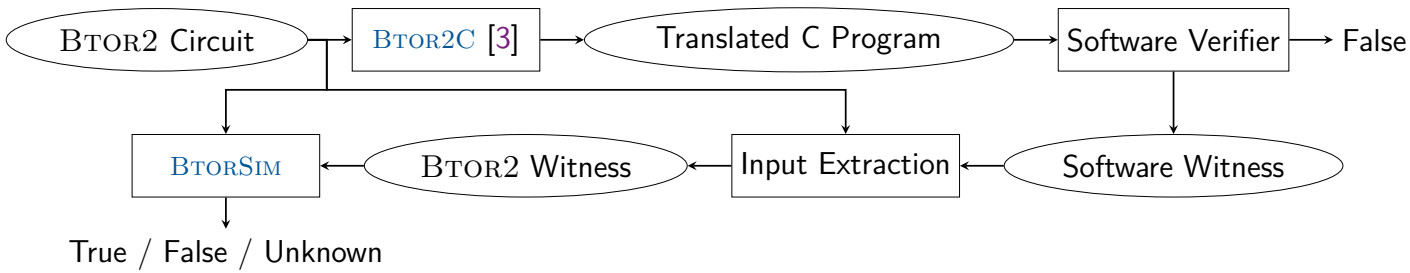

- Extract input assignments (and state initializations)
- Write a BTOR2 violation witness

# Violation Witnesses



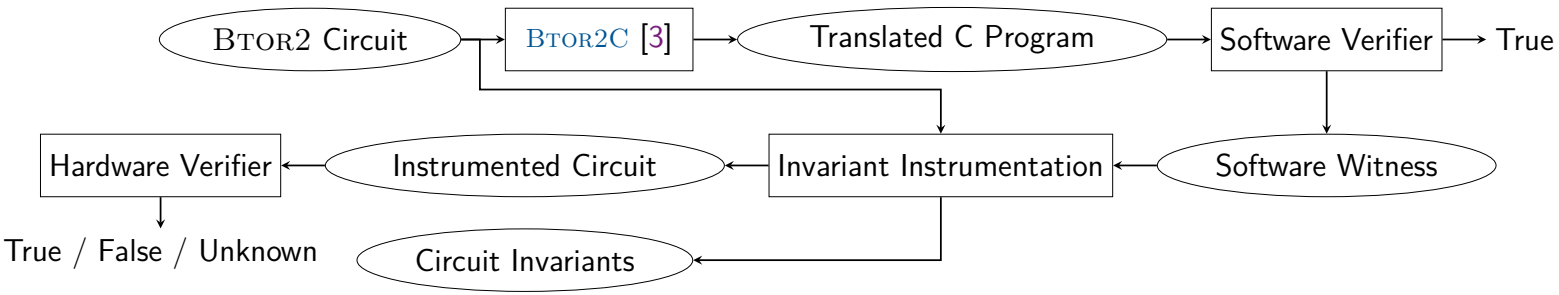

- Extract input assignments (and state initializations)
- Write a BTOR2 violation witness
- Simulate the circuit with BTORSIM [23]

# Violation Witnesses

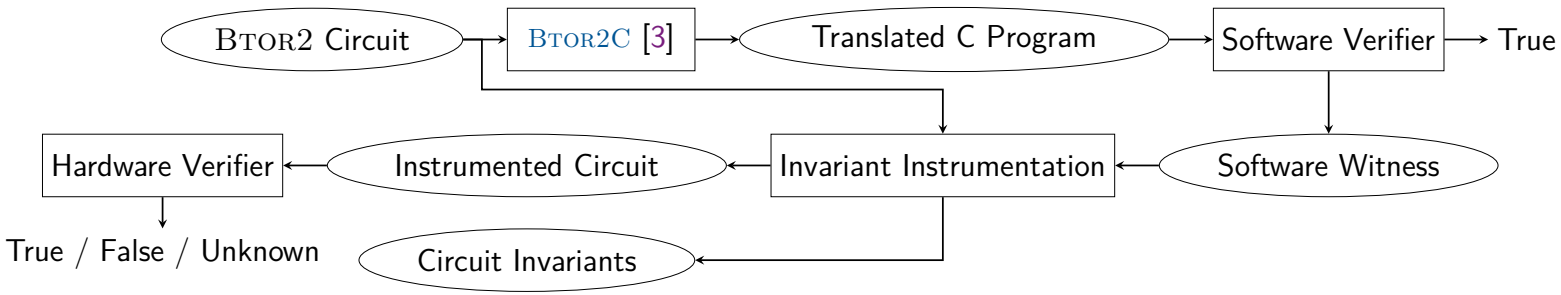



- Extract input assignments (and state initializations)
- Write a BTOR2 violation witness
- Simulate the circuit with BTORSIM [23]
  - Execution-based validation [4]

# Correctness Witnesses

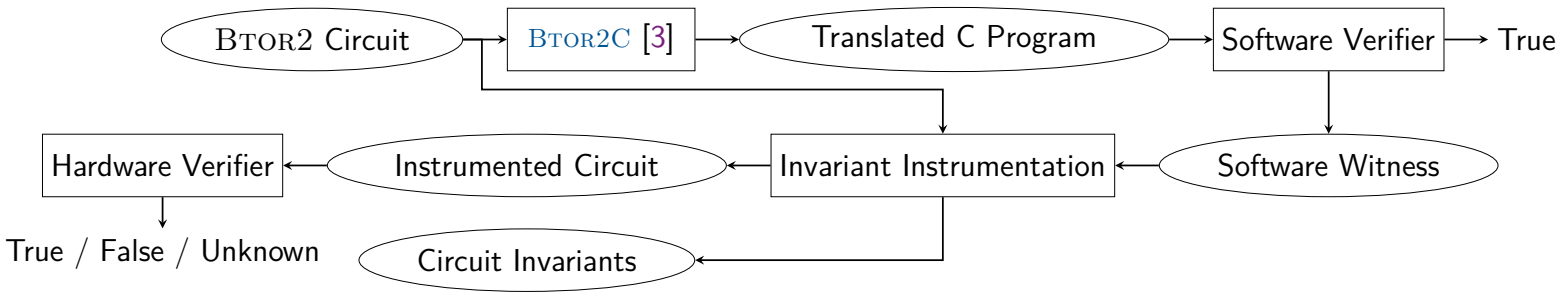


# Correctness Witnesses



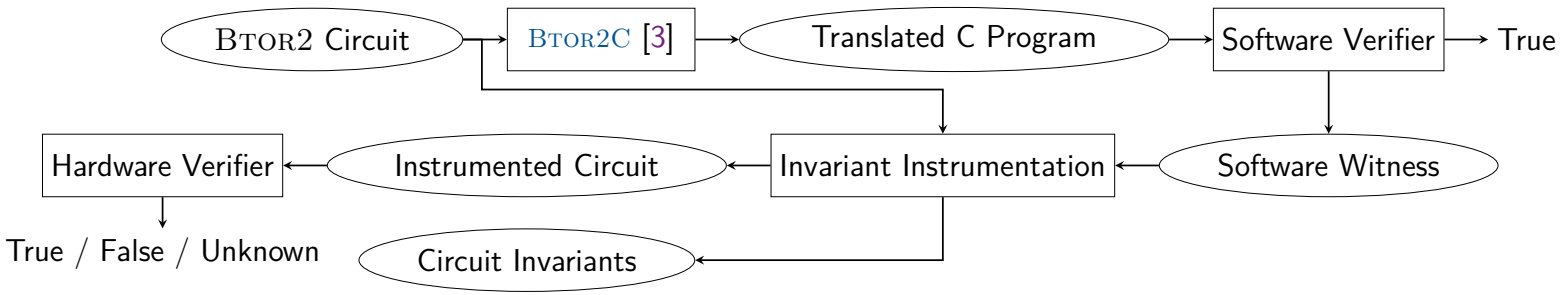

- BTOR2 has no format for correctness witnesses!

# Correctness Witnesses




- BTOR2 has no format for correctness witnesses!
- Validation via verification [6]

# Correctness Witnesses




- BTOR2 has no format for correctness witnesses!
- Validation via verification [6]
- Use hardware verifiers to validate software invariants

# What Invariants Should We Accept?

- Hardware model checking
  - $I(s)$: initial states
  - $T(s, s')$: transition relation
  - $P(s)$: safety property

# What Invariants Should We Accept?

- Hardware model checking
  - $I(s)$: initial states
  - $T(s, s')$: transition relation
  - $P(s)$: safety property
  - Goal: $R(s) \Rightarrow P(s)$

# What Invariants Should We Accept?

- Hardware model checking
  - $I(s)$: initial states
  - $T(s, s')$: transition relation
  - $P(s)$: safety property
  - Goal: $R(s) \Rightarrow P(s)$
- Invariants: $R(s) \Rightarrow Inv(s)$

# What Invariants Should We Accept?

- Hardware model checking
  - $I(s)$: initial states
  - $T(s, s')$: transition relation
  - $P(s)$: safety property
  - Goal: $R(s) \Rightarrow P(s)$
- Invariants: $R(s) \Rightarrow Inv(s)$
  - Accept trivial invariants (e.g., $\top$): re-verification

# What Invariants Should We Accept?

- Hardware model checking
  - $I(s)$: initial states
  - $T(s, s')$: transition relation
  - $P(s)$: safety property
  - Goal: $R(s) \Rightarrow P(s)$
- Invariants: $R(s) \Rightarrow Inv(s)$
  - Accept trivial invariants (e.g., $\top$): re-verification
- Safe invariants: $(R(s) \Rightarrow Inv(s)) \wedge (Inv(s) \Rightarrow P(s))$

# What Invariants Should We Accept?

- Hardware model checking
  - $I(s)$: initial states
  - $T(s, s')$: transition relation
  - $P(s)$: safety property
  - Goal: $R(s) \Rightarrow P(s)$
- Invariants: $R(s) \Rightarrow Inv(s)$
  - Accept trivial invariants (e.g., $\top$): re-verification
- Safe invariants: $(R(s) \Rightarrow Inv(s)) \wedge (Inv(s) \Rightarrow P(s))$
  - Reject trivial invariants
  - Invariants related to property: re-establish proofs

# What Invariants Should We Accept?

- Hardware model checking
  - $I(s)$: initial states
  - $T(s, s')$: transition relation
  - $P(s)$: safety property
  - Goal: $R(s) \Rightarrow P(s)$
- Invariants: $R(s) \Rightarrow Inv(s)$
  - Accept trivial invariants (e.g., $\top$): re-verification
- Safe invariants: $(R(s) \Rightarrow Inv(s)) \wedge (Inv(s) \Rightarrow P(s))$
  - Reject trivial invariants
  - Invariants related to property: re-establish proofs
- Safe and inductive invariants

# What Invariants Should We Accept?

- Hardware model checking
  - $I(s)$: initial states
  - $T(s, s')$: transition relation
  - $P(s)$: safety property
  - Goal: $R(s) \Rightarrow P(s)$
- Invariants: $R(s) \Rightarrow Inv(s)$
  - Accept trivial invariants (e.g., $\top$): re-verification
- Safe invariants: $(R(s) \Rightarrow Inv(s)) \wedge (Inv(s) \Rightarrow P(s))$
  - Reject trivial invariants
  - Invariants related to property: re-establish proofs
- Safe and inductive invariants
  - Pure SAT solving (do not need model checkers)
  - Most mature HW algorithms: IMC [21], PDR [9], etc.

# What Invariants Should We Accept?

- Hardware model checking
  - $I(s)$: initial states
  - $T(s, s')$: transition relation
  - $P(s)$: safety property
  - Goal: $R(s) \Rightarrow P(s)$
- Invariants: $R(s) \Rightarrow Inv(s)$
  - Accept trivial invariants (e.g., $\top$): re-verification
- Safe invariants: $(R(s) \Rightarrow Inv(s)) \wedge (Inv(s) \Rightarrow P(s))$
  - Reject trivial invariants
  - Invariants related to property: re-establish proofs
- Safe and inductive invariants
  - Pure SAT solving (do not need model checkers)
  - Most mature HW algorithms: IMC [21], PDR [9], etc.

The reason BTOR2 has no format for correctness witnesses?

# Preliminary Results on Correctness Witnesses

- Internship project of Zsófia Ádám from Budapest University of Technology and Economics

# Preliminary Results on Correctness Witnesses

- Internship project of Zsófia Ádám from Budapest University of Technology and Economics
- Implemented method $R(s) \Rightarrow Inv(s)$
- Evaluated CPAchecker (tried UAutomizer and 2LS)

# Preliminary Results on Correctness Witnesses

- Internship project of Zsófia Ádám from Budapest University of Technology and Economics
- Implemented method $R(s) \Rightarrow Inv(s)$
- Evaluated CPAchecker (tried UAutomizer and 2ls)
- Statistics
  - 867 safe hardware-verification programs (bit-vectors only)
  - 165 proved by predicate abstraction in CPAchecker
  - 114 out of 165 have no invariants in witnesses
  - 51 nontrivial invariants: 50 confirmed by AVR

# Preliminary Results on Correctness Witnesses

- Internship project of Zsófia Ádám from Budapest University of Technology and Economics
- Implemented method $R(s) \Rightarrow Inv(s)$
- Evaluated CPAchecker (tried UAutomizer and 2LS)
- Statistics
  - 867 safe hardware-verification programs (bit-vectors only)
  - 165 proved by predicate abstraction in CPAchecker
  - 114 out of 165 have no invariants in witnesses
  - 51 nontrivial invariants: 50 confirmed by AVR
- Working on stricter checks

# Reflections on Witness Formats for Programs

# Reflections on Witness Formats for Programs

- Violation witnesses: test cases
  - In BTOR2, a violation witness is a test pattern

# Reflections on Witness Formats for Programs

- Violation witnesses: test cases
  - In BTOR2, a violation witness is a test pattern
- Correctness witnesses: invariants

# Reflections on Witness Formats for Programs

- Violation witnesses: test cases
  - In BTOR2, a violation witness is a test pattern
- Correctness witnesses: invariants
  - McMillan, CAV 2006 [22] (IMPACT): "*An inductive invariant of a program is a map $I$, such that $I(l_i) = \top$ and for every action $(l, T, m)$, $I(l) \wedge T \Rightarrow I(m)$. A safety invariant of a program is an inductive invariant such that $I(l_f) = \bot$. Existence of a safety invariant of a program implies that the program is safe.*"

# Reflections on Witness Formats for Programs

- Violation witnesses: test cases
  - In BTOR2, a violation witness is a test pattern
- Correctness witnesses: invariants
  - McMillan, CAV 2006 [22] (IMPACT): "*An inductive invariant of a program is a map $I$, such that $I(l_i) = \top$ and for every action $(l, T, m)$, $I(l) \wedge T \Rightarrow I(m)$. A safety invariant of a program is an inductive invariant such that $I(l_f) = \bot$. Existence of a safety invariant of a program implies that the program is safe.*"
  - Require inductiveness: simplify validation!

# Reflections on Witness Formats for Programs

- Violation witnesses: test cases
  - In Btor2, a violation witness is a test pattern
- Correctness witnesses: invariants
  - McMillan, CAV 2006 [22] (Impact): "*An inductive invariant of a program is a map $I$, such that $I(l_i) = \top$ and for every action $(l, T, m)$, $I(l) \wedge T \Rightarrow I(m)$. A safety invariant of a program is an inductive invariant such that $I(l_f) = \bot$. Existence of a safety invariant of a program implies that the program is safe.*"
  - Require inductiveness: simplify validation!
- Automata might be too expressive for witnesses

# Reflections on Witness Formats for Programs

- Violation witnesses: test cases
  - In BTOR2, a violation witness is a test pattern
- Correctness witnesses: invariants
  - McMillan, CAV 2006 [22] (IMPACT): "*An inductive invariant of a program is a map $I$, such that $I(l_i) = \top$ and for every action $(l, T, m)$, $I(l) \wedge T \Rightarrow I(m)$. A safety invariant of a program is an inductive invariant such that $I(l_f) = \bot$. Existence of a safety invariant of a program implies that the program is safe.*"
  - Require inductiveness: simplify validation!
- Automata might be too expressive for witnesses
  - Violation: test case
  - Correctness: mapping from locations to invariants

# Conclusion

- Witnesses exchange information between HW/SW

# Conclusion

- Witnesses exchange information between HW/SW
- Certifying Btor2C by witness validation
  - Violation: input extraction + simulation
  - Correctness: invariant instrumentation + verification

# Conclusion

- Witnesses exchange information between HW/SW
- Certifying Btor2C by witness validation
  - Violation: input extraction + simulation
  - Correctness: invariant instrumentation + verification
- Strictness of correctness-witness validation
  - Invariant: may re-verify tasks
  - Safe invariants: can re-establish proofs
  - Safe and inductive invariants: simplify validation

# Conclusion

- Witnesses exchange information between HW/SW
- Certifying Btor2C by witness validation
  - Violation: input extraction + simulation
  - Correctness: invariant instrumentation + verification
- Strictness of correctness-witness validation
  - Invariant: may re-verify tasks
  - Safe invariants: can re-establish proofs
  - Safe and inductive invariants: simplify validation
- Reflections on the format
  - Violation: test case
  - Correctness: annotation (line $\mapsto$ invariant)

# References I


[1] IEEE Standard for Verilog Hardware Description Language (2006). https://doi.org/10.1109/IEEESTD.2006.99495

[2] 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

[3] Beyer, D., Chien, P.C., Lee, N.Z.: **Bridging Hardware and Software Analysis with Btor2C: A Word-Level-Circuit-to-C Translator**. In: Proc. TACAS. pp. 1–21. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_12

[4] Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1

[5] 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

[6] Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10

# References II


[7] 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

[8] Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009), isbn: 978-1-58603-929-5

[9] 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

[10] 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

[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. CAV. pp. 334–342. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22

[12] Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7

# References III


[13] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV. pp. 154–169. LNCS 1855, Springer (2000). https://doi.org/10.1007/10722167_15

[14] Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593

[15] 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

[16] 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

[17] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV. pp. 343–361. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20

[18] Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2

# References IV


[19] 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

[20] Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO. pp. 75–88. IEEE (2004). https://doi.org/10.1109/CGO.2004.1281665

[21] 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

[22] McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_14

[23] 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

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