

Benchmarking Software Model Checkers on Automotive Code

NFM 2020

Lukas Westhofen, Philipp Berger and Joost-Pieter Katoen

May 2020

Chair i2, RWTH Aachen University

Motivation

Software Model Checking

- very active field of research
- interest from industry is slowly mounting
- applicability, interoperability and stability is/was brittle
- enter the *Competition on Software Verification* (**SV-COMP**): from 9 tools in 2012 to >30 in 2019:
the most prestigious software verification competition!

Project History

- two year project with Ford Motor Company¹
- feasibility study: Model checking of automotive code
- two open-loop controller models as case studies
- previous subject of interest: *BTC EmbeddedValidator*, a commercial model checker
- **Outcome:** Feasible, but improvements are possible!



¹Berger, P., Katoen, J.P., Ábrahám, E., Waez, M.T.B., Rambow, T.: Verifying Auto-generated C Code from Simulink. In: FM. Volume 10951 of LNCS. (2018)

Our Questions

- How do the SV-COMP competitors perform on industrial, automotive code?
- How do these tools compare to proprietary tools that are tailored to such code?

Case Studies and Tools

The Case Studies

Basis: Two automotive case studies (open-loop controllers)

Electronic Clutch Control & Driveline State Request

- **Electronic clutch:**
replaces the manual shaft coupling
- *ECC* enables access to the electronic clutch
- **Driveline:** everything responsible for delivering power to the road
- *DSR* signalizes and sets driveline's state



~2500 LOC



~1350 LOC

Code Structure – General

```
1 // Global variables are declared here.
2 int motor_rpm;
3 extern float module_accl_paddle;
4
5 void initialize() {
6     // Initializes global variables.
7     motor_rpm = 2500;
8 }
9
10 void step() {
11     // Monolithic code for one bounded step.
12     motor_rpm *= module_accl_paddle;
13 }
14
15 // Entry point.
16 void main() {
17     initialize();
18     // Executes the step indefinitely.
19     while(1) {
20         step();
21     }
22 }
```


Verifier Selection

Three criteria for our use case:

- 1 Has a license that allows an academic evaluation
- 2 Operates on the features of the case studies
Rationale: Precise results
- 3 Competes in the categories *ReachSafety* and *SoftwareSystems* of the SV-COMP
Rationale: Maturity, applicability, SV-COMP functions

Verifier Selection

C-code model checkers

CBMC

SMACK

ESBMC

SYMBIOTIC

2LS

ULTIMATEAUTOMIZER

CPACHECKER

ULTIMATEKOJAK

PESCo

ULTIMATETAIPAN

DEPTHK

Verifier Selection

C-code model checkers

CBMC

SMACK

ESBMC

SYMBIOTIC

2LS

ULTIMATEAUTOMIZER

CPACHECKER

ULTIMATEKOJAK

PESCo

ULTIMATETAIPAN

DEPTHK

CBMC + k

CBMC + k

Enhancing BMC-only verifiers via k -induction*

- **Operation:** Code transformation that represents the induction step
- **Configurable:** Enables k -induction* on top of any bounded model checker
- **Leveraging:** Leverages efficiency of BMC-only verifiers for proof generation

***Specialized:** Works only on *this specific code structure!*

CBMC + k

k-induction code transformation

```
1  extern void __VERIFIER_error();
2
3
4  int main() {
5      initialize();
6
7
8      while(1) {
9
10
11         step();
12         if(!property())
13             __VERIFIER_error();
14     }
15 }
```

CBMC + k

k-induction code transformation

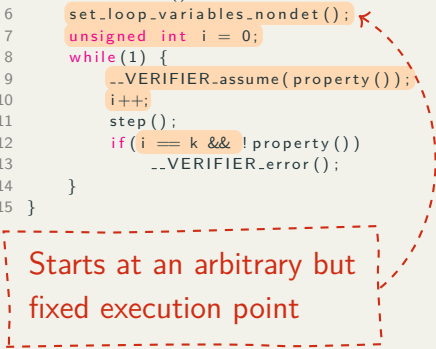
```
1  extern void __VERIFIER_error();
2  extern void __VERIFIER_assume(int);
3
4  int main() {
5      initialize();
6      set_loop_variables_nondet();
7      unsigned int i = 0;
8      while(1) {
9          __VERIFIER_assume(property());
10         i++;
11         step();
12         if(i == k && !property())
13             __VERIFIER_error();
14     }
15 }
```

$$IND_k(s_0, \dots, s_k) = \left(\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left(\bigwedge_{i=0}^{k-1} P(s_i) \right) \wedge \neg P(s_k)$$

CBMC + k

k-induction code transformation

```
1  extern void __VERIFIER_error();
2  extern void __VERIFIER_assume(int);
3
4  int main() {
5      initialize();
6      set_loop_variables_nondet();
7      unsigned int i = 0;
8      while(1) {
9          __VERIFIER_assume(property());
10         i++;
11         step();
12         if(i == k && !property())
13             __VERIFIER_error();
14     }
15 }
```

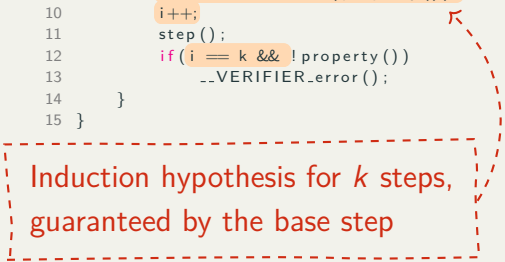


Starts at an arbitrary but
fixed execution point

CBMC + k

k -induction code transformation

```
1  extern void __VERIFIER_error();
2  extern void __VERIFIER_assume(int);
3
4  int main() {
5      initialize();
6      set_loop_variables_nondet();
7      unsigned int i = 0;
8      while(1) {
9          __VERIFIER_assume(property());
10         i++;
11         step();
12         if(i == k && !property())
13             __VERIFIER_error();
14     }
15 }
```



Induction hypothesis for k steps,
guaranteed by the base step

CBMC + k

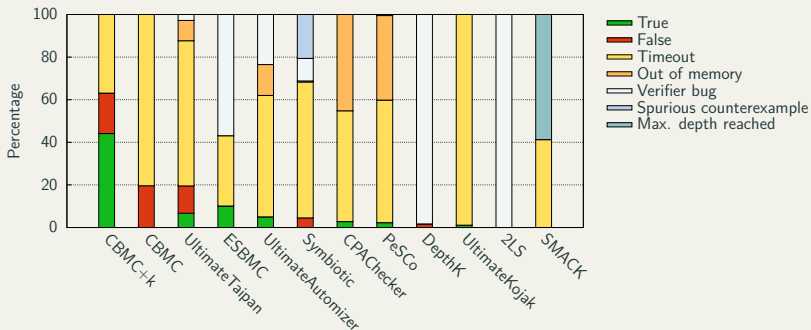
k -induction code transformation

```
1  extern void __VERIFIER_error();
2  extern void __VERIFIER_assume(int);
3
4  int main() {
5      initialize();
6      set_loop_variables_nondet();
7      unsigned int i = 0;
8      while(1) {
9          __VERIFIER_assume(property());
10         i++;
11         step();
12         if(i == k && !property()) ←
13             __VERIFIER_error();
14     }
15 }
```

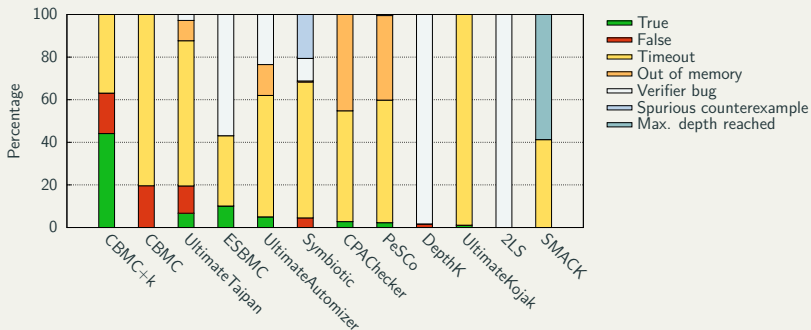
Checks if hypothesis was sufficient
for proof in iteration $k + 1$

Benchmarking

Overall – Result Distribution

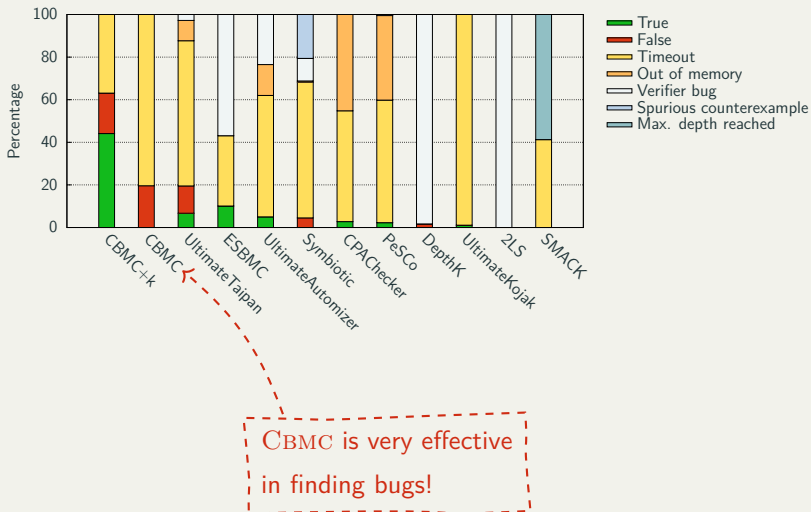


Overall – Result Distribution

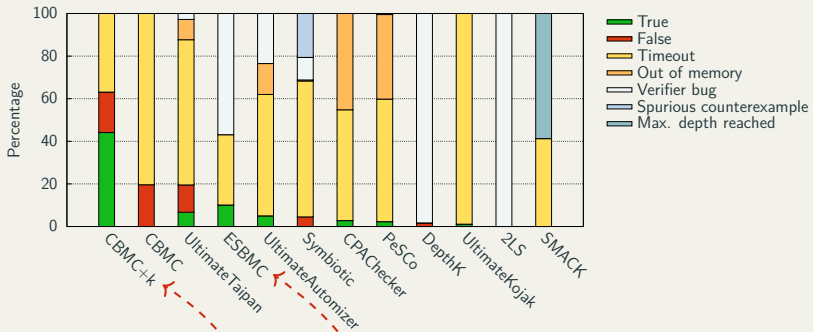


179 properties — more than 97% invariants!

Overall – Result Distribution

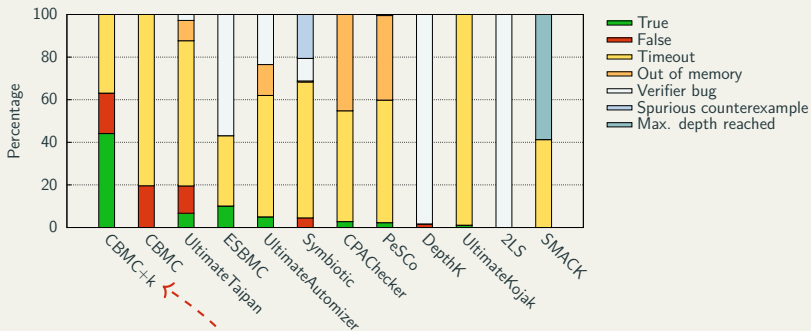


Overall – Result Distribution



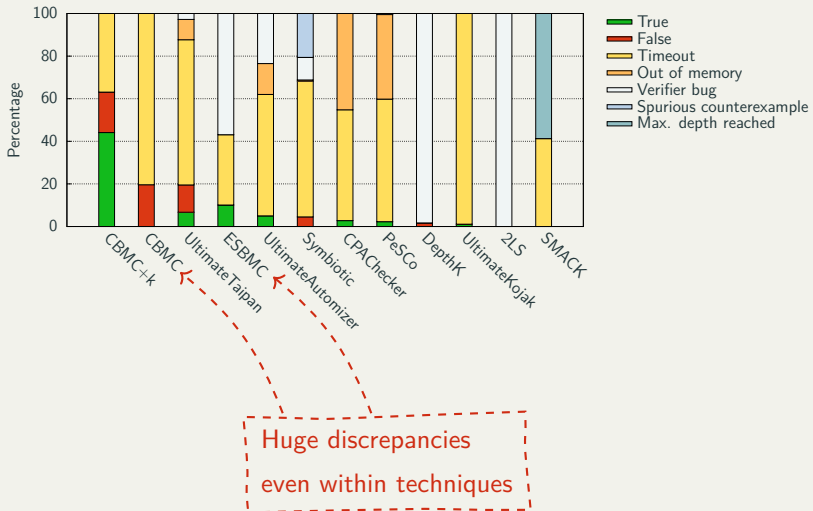
k-induction is the best
proof generation technique

Overall – Result Distribution

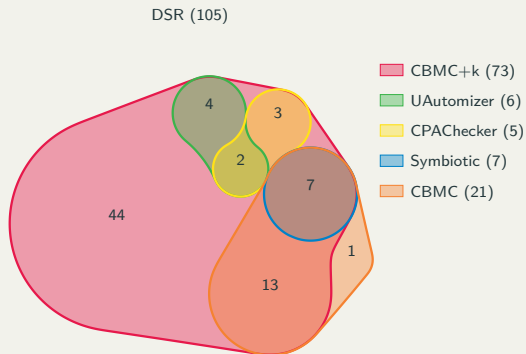


Leveraging CBMC with
k-induction works!

Overall – Result Distribution



Overall – Result Distribution in DSR



Most tools solve different problems – there are no easy ones!

Comparison to an Industrial Tool

Competitions like SV-COMP use a *ground truth* for assigning scores for **correct** and **incorrect** answers.

Verification result	<i>False</i>			<i>True</i>		
Validation result	✓	?	✗	✓	?	✗
Score	+1	±0	±0	+2	+1	±0

Comparison to an Industrial Tool – BTC

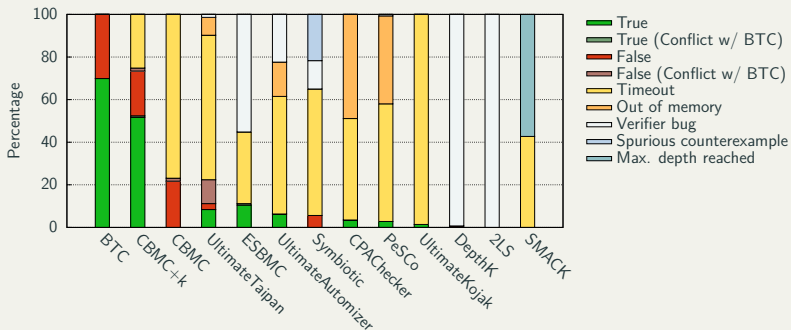
BTC EmbeddedValidator (BTC)

- is focused on embedded automotive C-code, but
- can not (easily) handle more general code, and
- was run on slower CPU and with less RAM.

So why rely on BTC? It is

- a mature, commercial tool,
- *specialized* to this use case, and
- provides good coverage on the case studies (143 of 179).

Comparison to an Industrial Tool – BTC



The verification results for each verifier, in percent of the 143 verification tasks on which BTC returned a definite result.

Epilogue

Our Answers

For the examined use case...

- How do the SV-COMP competitors perform on industrial, automotive code?

*There seems to be
a serious gap between
the needs of automotive code verification
and open-source software model checker capabilities.*

At most 20% coverage on global invariants!

Our Answers

For the examined use case...

- How do these tools compare to proprietary tools that are tailored to such code?

Quantitative Results: To be expected.

Qualitative Results: *Surprisingly* bad!

But: Applicability should come in academic focus!

Main Takeaways

- **More Benchmarks.** Industrial partners need to come forward with more real-world case studies not entangled in NDAs.
- The **scoring scheme** in SV-COMP. The punishment of wrong verification results is too severe! A relative judgment (% of wrong answers) seems to be more fair.

Code Structure – Specifics

Metric	ECC	DSR
<i>Source lines of code</i>	2517	1354
<i>Global constants</i>	274	77
float	30%	58%
<i>Global variables</i>	775	273
float	23%	26%
<i>Operations</i>	10096	5232
Addition/subtraction	346	133
Multiplication/division	253	52
Bit-precise operations	191	65
Pointer dereferences	180	83
...		

Reasons for bad coverage

- Exploiting the code structure is key
- Preprocessing and handling for pointer-magic and bitmask-on-float
- Access to industrial code for testing and adapting

Detailed – Verifier Stability

11 issues encountered during the study

CBMC: 2

- Incorrect handling of switch-local variables (✓)
- Faulty witness format (✓)

ESBMC/DEPTHK: 1

- Faulty SMT formula for Boolector

2LS: 2

- False negatives with standard configuration
- Bug in bit-vector implementation

CPACHECKER: 3

- Resolving typedef's (✓)
- Ignoring of switch-local variables
- Incomplete implementation of Z3 glue code

UAUTOMIZER/UTAIPAN: 2

- Conversion error of an assertion
- Program abortion through unknown enum

SYMBIOTIC: 1

- Fails verification due to KLEE shortcomings

Contradicting Results

The contradicting results observed in DSR and ECC, respectively.

Case study	True	False
<i>DSR</i>	CBMC+k	BTC
	BTC	CBMC, CBMC+k
<i>ECC</i>	BTC	UltimateTaipan
	BTC, CBMC+k	UltimateTaipan
	BTC, ESBMC, CBMC+k	UltimateTaipan
	BTC, ESBMC	UltimateTaipan
	ESBMC, CBMC+k	DepthK
	ESBMC	BTC, UltimateTaipan

References i



Berger, P., Katoen, J.P., Ábrahám, E., Waez, M.T.B.,
Rambow, T.:

Verifying Auto-generated C Code from Simulink.

In: FM. Volume 10951 of LNCS. (2018) 312–328