Benchmarking Software Model Checkers on Automotive Code

Lukas Westhofen[1], Philipp Berger[2(✉)], and Joost-Pieter Katoen[2]

[1] OFFIS e.V, Oldenburg, Germany
lukas.westhofen@offis.de
[2] RWTH Aachen University, Aachen, Germany
{berger,katoen}@cs.rwth-aachen.de

Abstract. This paper reports on our experiences with verifying automotive C code by state-of-the-art open source software model checkers. The embedded C code is automatically generated from Simulink open-loop controller models. Its diverse features (decision logic, floating-point and pointer arithmetic, rate limiters and state-flow systems) and the extensive use of floating-point variables make verifying the code highly challenging. Our study reveals large discrepancies in coverage—which is at most only 20% of all requirements—and tool strength compared to results from the main annual software verification competition. A hand-crafted, simple extension of the verifier CBMC with k-induction delivers results on 63% of the requirements while the proprietary BTC EmbeddedValidator covers 80% and obtains bounded verification results for most of the remaining requirements.

1 Introduction

Software Model Checking. Software model checking is an active field of research. Whereas model checking algorithms initially focused on verifying models, various dedicated techniques have been developed in the last two decades to enable model checking of program code. This includes e.g., predicate abstraction, abstract interpretation, bounded model checking, counterexample-guided abstraction refinement (CEGAR) and automata-based techniques. Combined with the enormous advancements of SAT and SMT-techniques [1], nowadays program code can be directly verified by powerful tools. Companies like Microsoft, Facebook, Amazon, and ARM check software on a daily basis using in-house model checkers. The enormous variety of code verification techniques and tools has initiated a number of software verification competitions such as RERS, VerifyThis, and SV-COMP. For software model checking, the annual SV-COMP competition is most relevant. Launched with 9 participating tools in 2012, it gained popularity over the years with more than 40 competitors in 2019 [2]. It runs off-line in a controlled manner, and has several categories. Competitions like SV-COMP have established standards in input and output format, and evaluation criteria. Software model checkers are ranked based on the verification results, earning points for correct results while being punished for wrong outcomes. A more recent

R. Lee et al. (Eds.): NFM 2020, LNCS 12229, pp. 133–150, 2020.
https://doi.org/10.1007/978-3-030-55754-6_8

development is the usage of witnesses to validate verification results. Results are provided in so-called quantile plots indicating the required verification time versus the cumulative score over the benchmarks.

Aims of this Paper. This paper focuses on: *how do the SV-COMP competitors perform on automotive code?* and *how do these tools compare to proprietary tools that are tailored to such code?* The objective of this paper is to benchmark a rich set of participating tools in SV-COMP using two case studies from a major car manufacturer taken from [3]. In contrast to the SV-COMP, where a diverse set of open-source verification tasks ranging from small academic examples over concurrent programs up to software systems are submitted by research and development groups, we focus on an industrial grade automotive code base. To the best of our knowledge, such an evaluation has not been made before. While a set of two case studies is certainly a small benchmark in comparison, the size of the two case studies (of about 1400 and 2500 lines of embedded C code respectively), its diverse features (decision logic, floating-point arithmetic, pointer dereferencing, rate limiters, bitwise operations and state-flow systems), the rich set of (179) requirements, and the availability of verification results obtained by the proprietary software model checker BTC EmbeddedValidator, make it an interesting starting point to validate and compare various open-source software model checkers on an automotive code base.

Approach. We selected 11 software model checkers from the SV-COMP 2019 [2], based on (a) the aforementioned characteristics of the two automotive case studies, (b) the requirements that mostly are safety properties, and (c) the availability of a license that enables an academic evaluation. In addition, we considered a simple hand-crafted extension of CBMC [4] with k-induction that is tailored to the control-flow characteristics of the two benchmarks. We conducted two main experiments. The first experiment runs the 12 software model checkers on the 179 requirements, 99% of which are invariants, and *focuses on comparing the coverage of the tools (how many requirements could be verified or refuted), and their verification time*. The second experiment *benchmarks the open-source code verifiers against the proprietary verifier BTC EmbeddedValidator*[1].

Our Main Findings. The main results of this paper are:

- The SV-COMP competitors are able to obtain results for at most 20% of all requirements. Various competitors covered between 0 and 5% only.
- A hand-crafted, simple extension of CBMC with k-induction covers 63%.
- BTC EmbeddedValidator covers 80% and obtains bounded verification results for 85% of the remaining requirements.

Our results show that there is a lot of untapped optimization potential for making existing open source software model checkers more appealing and applicable to automotive code. Suitable benchmark candidates are currently too closely

[1] https://www.btc-es.de/en/products/btc-embeddedplatform/.

guarded by industry to be really driving scientific development. Therefore, the message of this paper is to emphasize the need for a synchronization between the industrial and scientific software verification communities.

2 Preliminaries

2.1 The Automotive Benchmarks

Benchmark Description. Both case studies involve auto-generated code of two R&D prototype Simulink models from Ford Motor Company: the next-gen *Driveline State Request* (DSR) feature and the next-gen *E-Clutch Control* (ECC) feature. The DSR and ECC features implement the decision logic for opening and closing the driveline and calculating the desired clutch torque and corresponding engine control torque of the vehicle, respectively. The case studies are described in detail in [3]. Unfortunately, because of non-disclosure agreements, we cannot make the benchmarks publicly available; instead we give a detailed characterization of the used code in the following.

Code Characteristics. From the Simulink models, generated by a few thousand blocks, around 1,400 and 2,500 source lines of C code were extracted for DSR and ECC. Both code bases have a cyclomatic complexity of over 200 program paths. The cyclomatic complexity is a common software metric indicating the number of linearly independent paths through a program's code. Table 1 presents the metrics collected on both case studies.

Table 1. Code metrics of the benchmarks.

Metric	DSR	ECC
Complexity		
Source lines of code	1,354	2,517
Cyclomatic complexity	213	268
Global constants	*77*	*274*
`char`	12	8
`char[]`	`[12,32]` 2	0
`float`	35	77
`float[]`	`[6-12]` 9	`[2-7]` 4
`float*`	1	1
`void*`	18	184
Global variables	*273*	*775*
`char`	199	595
`char[]`	`[16-32]` 3	0
`float`	46	110
`float[]`	`[4-10]` 25	`[2-4]` 70
Operations	*5232*	*10096*
Addition/subtraction	133	346
Multiplication/division	52	253
Bit-wise operations	65	191
Pointer dereferences	83	180

Constants are used to account for configurability, i.e. they represent parameters of the model that can be changed for different types of applications. The configurable state-space consists of 77 and 274 constants, for DSR and ECC respectively. Most of them are of type `float`, sometimes in a fixed-length array, as indicated by the square brackets. Their size range is also given in square brackets. Additionally, both case studies contain pointers to constant data (e.g. `const void*`).

With a couple of hundred variables, *globals are heavily employed*. They are used for exchanging data with other compilation units. Here, the `char` type is

most prevalent, taking up around three quarters of the variable count. `float` variables make up the remaining quarter.

The number of operations in the call graph are around 5,000 and 10,000 for DSR and ECC. While *linear arithmetic* is most prominent, we also observe a large amount of *multiplication* and *division* operations, possibly on non-constant variables. Challenges for software verifiers rise along with the complexity of operators used. Pointer and floating-point arithmetic, as well as bit-wise operations impose challenges. These case studies employ a variety of *bit-wise operations* such as `>>`, `&`, and `|`, mainly on 32-bit variables. Such operators can force the underlying solvers to model the variable bit by bit. A noticeable amount of *pointer dereferences*, namely 180 and 83 occurrences, is present in the programs.

Requirement Characteristics. The requirements originate from internal and informal documents of the car manufacturer and have been formalized by hand. As described in [3], obtaining an unambiguous formal requirement specification can be a substantial task. All differences between the formalization in [3] and this work in number of properties stem from different splitting of the properties. For the DSR case study, from 42 functional requirements we extracted 105 properties, consisting of 103 invariants and two bounded-response properties. For the ECC case study, from 74 functional requirements we extracted 71 invariants and three bounded-response properties.

Invariant properties are assertions that are supposed to hold for all reachable states. Bounded-response properties request that a certain assertion holds within a given number of computational steps whenever a given, second assertion holds.

2.2 The Software Model Checkers

In order to analyze the performance of open-source verifiers on our specific use case of embedded automotive C code from Simulink models, we selected a suitable subset of C verifiers based on the following criteria:

1. Has matured enough to compete in the SV-COMP 2019 [2] in the *ReachSafety* and *SoftwareSystems* category.
2. Has a license that allows an academic evaluation.

Based on these criteria, we selected the verifiers: 2LS, CBMC, CPAChecker, DepthK, ESBMC, PeSCo, SMACK, Symbiotic, UltimateAutomizer, UltimateKojak, and UltimateTaipan. The study was conducted in March 2019. We used the latest stable versions of each tool to that date. We also included CBMC+k (described in Sect. 2.3), a variant of CBMC that enables k-induction as a proof generation technique on top of CBMC. Let us briefly introduce the selected open-source verifiers.

CBMC 5.11 [5]. The *C Bounded Model Checker* is a matured bounded model checker for C programs. CBMC takes a pre-specified bound up to which the

program loops are unrolled. The resulting transition system is encoded symbolically, and finally passed to an SAT-solver. For a given bound k, this formula over the program states is created in the following manner, where I is the initial condition, T the transition relation, s_i a state and P the property:

$$BMC_k(s_0, \ldots, s_k) = I(s_0) \wedge \left(\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left(\bigvee_{i=0}^{k} \neg P(s_i) \right) \tag{1}$$

ESBMC 6.0.0 [6]. The *Efficient SMT-based Bounded Model Checker* was forked off of a 2008 version of CBMC and has been replacing original framework parts ever since. One of its goals is to directly translate to SMT-theories instead of relying on SAT-solvers. It furthermore supports k-induction. Here, a generalized mathematical induction is applied to program loops, where a "look-back" of k steps is allowed for the induction hypothesis. The verification task can be specified as a formula over the program states:

$$IND_k(s_0, \ldots, s_k) = \left(\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left(\bigwedge_{i=0}^{k-1} P(s_i) \right) \wedge \neg P(s_k) \tag{2}$$

2LS 0.7.0 [7]. This is another fork of CBMC that expands from bounded model checking to a multitude of verification approaches. It interprets program analysis as a problem of solving a second-order logic instance. This leads to a variety of concepts that 2LS can employ, including (incremental) bounded model checking, k-induction, k-induction k-invariants, and abstract interpretation.

CPAChecker 1.8.0 [8]. The *Configurable Program Analysis Checker* provides a framework for implementing a rich set of analysis and verification techniques. By employing an abstract analysis algorithm, it implements concrete approaches such as predicate abstraction [9], value analysis [10], and k-induction [11].

PeSCo 1.7 [12]. PeSCo is a recent fork of CPAChecker which exploits *machine learning* to effectively select a fitting configuration for the given verification task.

DepthK 3.1 [13]. DepthK uses *k-induction on top of ESBMC* combined with an *invariant-strengthening* approach. It supports the iterative proof process by inferring possibly over-approximating invariants over polyhedral constraints.

SMACK 1.9.3 [14]. Rather than being a verifier by itself, SMACK translates from the LLVM intermediate representation (IR) into the *Boogie* [15] intermediate verification language (IVL). Corral, the default verification back end, employs bounded model checking with a goal-directed search algorithm.

Symbiotic 6.0.3 [16]. Symbiotic applies program instrumentation, *static slicing* and symbolic execution to identify counterexamples. Internally, it uses a patched KLEE version for symbolic execution and witness generation.

UltimateAutomizer 91b1670e [17]. This tool implements a trace-abstraction based on *automata* in a CEGAR fashion. Its development is based on the Ultimate framework which provides access to program representation, code transformations, and SMT-solvers. It applies a CEGAR scheme until an error automaton with sufficient abstraction is found.

UltimateKojak 91b1670e [18]. As part of the Ultimate tool chain, UltimateKojak uses CEGAR with *interpolation* over multiple program paths.

UltimateTaipan 91b1670e [19]. Similar to UltimateAutomizer, UltimateTaipan employs automata-based trace abstraction and CEGAR. It uses a fixed-point iteration to refine error paths until a sufficient precision is reached.

2.3 A Simple, Tailored Variant of CBMC

The SV-COMP verifiers are complemented by a simple, hand-crafted extension of the bounded model checker CBMC (version 5.11) with k-induction. Our implementation is tailored to the two case studies, in particular to programs with one main outermost control loop. Our prime motivation to consider this variant is to show the effect of a simple, almost trivial, tweak of a bounded model checker. The main goal of k-induction is harvesting the power of efficient bounded model checkers such as CBMC for proof generation. In this way, verifiers that natively only support bug hunting but have matured over time, can be elevated.

```
extern void __VERIFIER_error();

int main() {
  initialize();


  while(1) {


    step();
    if(!property())
      __VERIFIER_error();
  }
}
```

⇝

```
extern void __VERIFIER_error();
extern void __VERIFIER_assume(int);
int main() {
  initialize();
  set_loop_variables_nondet();
  unsigned int i = 0;
  while(1) {
    __VERIFIER_assume(property());
    i++;
    step();
    if(i == k && !property())
      __VERIFIER_error();
  }
}
```

Fig. 1. The transformation that is applied in the `k`-th induction step.

Our implementation CBMC+k is realized by a straightforward code transformation [20], see Fig. 1. It creates a new program representing the induction step such that all input variables are set non-deterministically on entering the loop. It then runs the back-end verifier on both the base step – i.e. the input file – and the induction step. If the base step returns a counterexample, the tool reports *False*. In case the induction step returns no counterexample for iteration k and the base case has also reached k, it reports *True*. Our two case studies do not require the forward case in [20], thus simplifying the implementation.

CBMC+k has *severe restrictions on its input code*. It is targeted to embedded C programs containing one (unbounded) main loop with a strictly bounded loop body. The property has to be checked at the very end of every loop iteration. Although there exist transformations from general programs to one-loop programs, we decided to skip this step as our case studies do not exhibit nested unbounded loops. Evidently, CBMC+k inherits the capabilities (and deficiencies) of its back-end verifier, specifically its ability to handle large state spaces.

CBMC+k should thus not be considered a generic, widely applicable extension of CBMC with k-induction. Our prime motivation to consider this variant is to show the effect of a simple, almost trivial, tweak on a bounded model checker. We have taken CBMC for this variant as it performed very well in identifying counterexamples, an important trait for k-induction.

The CBMC+k implementation is made publicly available at https://github.com/moves-rwth/cbmc-with-kInduction.

2.4 Experimental Setup

All experiments were performed on a machine with 192 GB RAM and two Intel Xeon Platinum 8160 processors, each containing 24 cores at 2.1 GHz. Our benchmark script executed ten benchmarks in parallel, giving each execution four CPU cores with a memory limit of 18 GB and a CPU-time limit of two hours. Further details can be found in the Appendix. Every verification was followed by two witness validation runs of CPAChecker and Ultimate. Conforming to the regulations of the SV-COMP 2019, the time limit for a correctness witness was two hours, whereas a violation witness had to complete within 12 min. We collected the data points:

- the result; either *True*, *False* or *Unknown*,
- why no definite answer was given, e.g. *Timeout*, *Memout* or *Verifier bug*,
- the used CPU-time, in seconds,
- the peak memory usage, in MB,
- if measurable, the time spent by a SAT solver, in seconds,
- if measurable, the reached depth in a BMC or k-induction setting, and
- the witness validation results; either *Correct* (validation result = original result), *Invalid* (unparseable witness) or *Unknown* (resource exhaustion, or validation result ≠ original result).

To keep the results comparable and the competition fair, we used the default configurations that the tool maintainers chose for the SV-COMP 2019 reachability tasks. The exact settings can be found in the Appendix. CBMC was invoked with increasing values of k by a wrapper script similar to the one employed in the SV-COMP 2019. For the Ultimate tool chain, a bit-precise memory model was applied[2] to the Boogie translator configuration. The witness validation processes for CPAChecker and Ultimate were set up as in SV-COMP 2019 with scaled run times where necessary. Due to the aforementioned confidentiality reasons, we cannot disclose the extracted benchmark data and verifier outputs.

[2] by adding `Memory model=HoenickeLindenmann_Original`.

3 Comparing the Open-Source Verifiers

Coverage. Figure 2 shows the verification results of running the open-source verifiers on the two case studies, omitting the results of the witness validation.

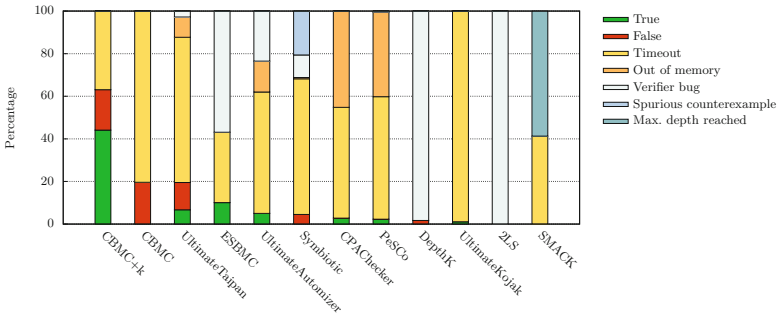

Fig. 2. The overall result distribution for each software model checker, in percent.

CBMC+k is able to verify about 63% of the verification tasks; CBMC and UltimateTaipan cover roughly 20%. ESBMC delivers results on 10% of the requirements. The remaining verifiers reach a coverage of at most 5%. The majority of the verifiers is either able to identify counterexamples or produce proofs, but seldom both. 2LS and SMACK cannot return a single definite result. The only successful witness validation was a proof of PeSCo validated by CPAChecker, indicated by *True (Correct)*. CBMC delivered invalid witnesses on all tasks, leading it to fail the witness validation process.

Figure 2 also indicates the reasons for *Unknown* answers. We observe that *time- and memory-outs prevail*, but a large number of verifiers exhibit *erroneous behavior*. A detailed description of the latter issues is given in Sect. 5.

To get insight into which requirements are covered by which software model checker, Fig. 3 depicts two Venn diagrams indicating the subsets of all 179 verification tasks. Each area represents the set of verification tasks on which a verifier returned a definite result. Those areas are further divided into overlapping subareas, where a number indicates the size of this set. For reasons of clarity, we included only the top five verifiers for the respective case study, based on the number of definite answers. For both case studies, there is not one verifier which covers all requirements covered by the other verifiers. For DSR, CBMC+k covers all but one definite results of the remaining verifiers. In this case, CBMC was able to identify a counterexample close to the timeout. CBMC+k exhausts its resources on this requirement as the inductive case occupies a part of the available computation time. For ECC, UltimateTaipan, ESBMC, and CBMC+k together cover the set of all definite results. Note that some verifiers—e.g. UltimateTaipan and ESBMC—perform rather well on one case study, but lose most of their coverage on the other. In most of such cases, this is due to erroneous behavior of the verifier manifesting on just one of the two case studies.

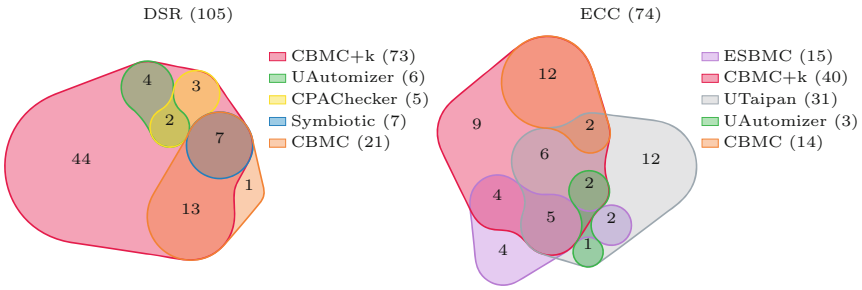

Fig. 3. Venn diagrams indicating the requirement coverage (i.e., a definite result was issued) by the top-five verifiers for case study DSR (left) and ECC (right).

We believe that the substantial difference in verifier coverage for the two case studies, as seen in Fig. 3, is the result of structural differences in the benchmark code. While the overall control-flow structure (closed loop, step-based input to output propagation) is the same for DSR and ECC, the difference in overall size and the higher number of global constants, pointers and floating-point variables make ECC imposing different challenges. Even a small increase in code size can lead to verifiers not even getting through costly initial preparatory steps, that, if completed, might have quickly been followed by a result.

Quantile Plot. As standard in SV-COMP, a quantile plot for the results on both case studies together is depicted in Fig. 4. Note the log-log scale. To this end, a score is assigned to each verification run according to the SV-COMP[3] scheme in Table 2.

Table 2. The employed scoring scheme for the quantile plots as adopted from SV-COMP.

Verification result	*False*			*True*		
Validation result	✓	?	✗	✓	?	✗
Score	+1	±0	±0	+2	+1	±0

The score depends on the results of the witness validation which can either be validated (verification and validation result coincide, indicated by ✓), not validated (resource exhaustion or verification and validation result differ, **?**) or invalid (unparseable witness, ✗). In absence of expected verification results, no punishments for wrong answers are given. In Sect. 4, we compare the verification results against those obtained by the commercial verifier BTC EmbeddedValidator.

[3] https://sv-comp.sosy-lab.org/2019/rules.php#scores.

The quantile plot in Fig. 4 indicates the accumulated score for all verification runs, sorted by ascending run time (x-axis), against the required CPU-time (y-axis). A (log, log) scale is used for improved readability. As invalid and unvalidated counterexamples are not rewarded in this score, verifiers returning such results – 2LS, CBMC, DepthK, SMACK and Symbiotic – obtain a zero score. Verifiers with a large number of proofs obtain higher scores. As only one witness could be validated, this aspect plays a negligible role in the scores. CBMC+k exhibits a higher score than other verifiers; runner-up ESBMC obtains various results only after one hour. In general, 50% and 90% of the answers were given within seven and 75 min, respectively. Only few verifiers used the full time limit of two hours: The Ultimate verifiers and CBMC+k obtained many results within an hour.

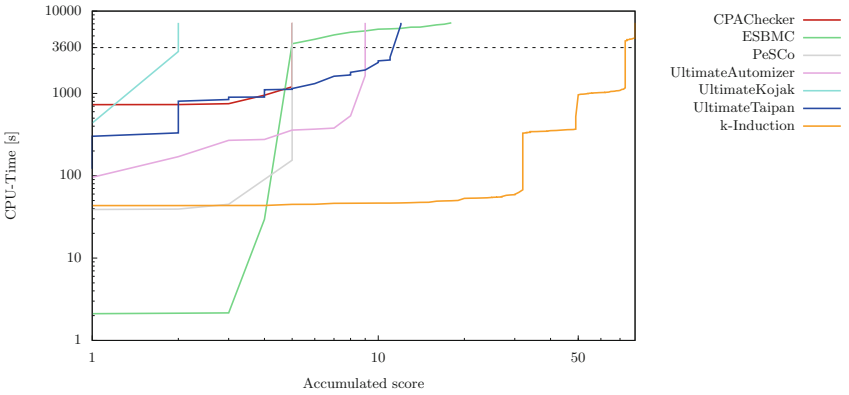

Fig. 4. The quantile (log, log) plot for all verifiers except the tools 2LS, CBMC, DepthK, SMACK and Symbiotic (as they reach a zero score).

4 Benchmarking Against BTC EmbeddedValidator

To compare the results of open-source software verifiers to a commercial tool, we additionally ran the verification tasks using BTC EmbeddedValidator (BTC for short).[4] The main purpose of this examination is the establishment of a reference point. This reference can subsequently be used as a foundation to interpret the applicability of the open-source verifiers to the industrial case studies.

BTC EmbeddedValidator is part of BTC EmbeddedPlatform, a commercial model-checking tool developed for industrial applications. It is, among others, heavily optimized for industrial embedded software—such as the benchmarks

[4] Similar results were provided in [3]. We have used a more recent version of BTC EmbeddedValidator and considered 179 rather than 112 requirements, as requirements were split differently.

considered in this paper—and unsurprisingly performs very well on the ECC and DSR case studies. This focus is also a weak point: It can not or not easily deal with memory allocation and many standard library headers usually not present in the targeted embedded code, making it unsuitable for a direct comparison on established SV-COMP benchmarks. Requirements can be specified directly using a pattern-based approach, see [21,22]. BTC EmbeddedValidator employs several back-end tools for verification: CBMC[5], iSAT3, AutoFXP, SMIBMC, and VIS. Code transformation, static analysis, and detection of spurious, i.e., incorrect, counterexamples are done as part of the verification.

We used BTC EmbeddedPlatform 2.3p1 under Windows 7 with 4 GB RAM and an Intel i7-6700HQ with a timeout of two hours. While this setup is using a smaller CPU and less RAM than our experiment in Sect. 3 and is therefore incomparable, it is important to stress that we use the results of BTC only for deciding the baseline truth and do not depend on the performance (see also Sect. 4.2).

4.1 BTC EmbeddedValidator Verification Results

Table 3 states the result distribution for both case studies, in percent of the 105 and 74 verification tasks, respectively.

Table 3. Verification results of BTC EmbeddedValidator on both case studies, in percent of the 105 and 74 verification tasks.

Case study	DSR (105)			ECC (74)		
Result	*True*	*False*	*Unknown*	*True*	*False*	*Unknown*
Percentage	56.2%	21.9%	21.9%	55.4%	27.0%	17.6%

BTC did not return a result on 21.9% of the DSR tasks; 91% of which were due to reaching only bounded correctness, but no unbounded proof. BTC timed out on the remaining 9%. Of the 17.6% *Unknown* answers for ECC, 92% are bounded proofs, and 8% timed out. In comparison to the open-source verifiers, BTC takes first place in both case studies when considering the overall number of definite answers. As witness output is not available in BTC and wall clock times were measured, we cannot integrate BTC fairly into our scoring system, and thus refrain from calculating a quantile plot score. Although we were not able to determine exact CPU-times from BTC due to tool limitations, a wall clock time was collected. The average wall clock time of BTC on tasks where definite answers were returned amounts to 17 ± 4 s on DSR and 308 ± 1109 s on ECC. Figure 5 shows the results for all 143 verification tasks on which BTC returned a definite result. It also indicates conflicts, i.e. different outcomes than BTC EmbeddedValidator.

[5] A different, custom version than used in SV-COMP 2019.

4.2 Scores Assuming Correct Results by BTC EmbeddedValidator

In absence of the true verification results, let us assume the results of BTC EmbeddedValidator as a "ground truth". As this is a mature industrial tool developed over many years specifically for such industrial cases considered here, we believe that this is a reasonable assumption. For this, we restrict the verification tasks to those on which BTC returns a definite answer. We are aware of the fact that this is debatable, but given the very low number of verification results by BTC EmbeddedValidator that could be shown by other tools to be invalidated (as depicted later), this gives a quite good impression. We would like to point out that we are not interested in either shaming or praising specific tools, we simply are trying to provide a look at the "big picture" with respect to model checking certain types of industrial embedded code. Our assumption of using BTC EmbeddedValidator as a ground truth does certainly not imply the validity of all its results. But, considering the purpose of this section, it represents a sufficiently precise reference point for a comparison. We update the quantile plots to *now punish wrong results* (i.e., results in conflict with BTC) by -16 *and* -32 *points for wrong violation and proof results*, respectively, as in the SV-COMP. The resulting plots are given in Fig. 6.

Compared to Fig. 4, the scores of CBMC+k are substantially worse as it has three conflicts with BTC EmbeddedValidator. This is due to the fact that the SV-COMP punishment scheme is bad for verifiers returning many results of which some are wrong. It is almost as good (in terms of the scoring scheme) to not generate any result at all (and thus no "wrong" result). This effect is certainly important when witness validaation is seldom, as it is the case in our setting where only one witness could be validated. CBMC+k produced definite verification results on many of the requirements, and consequently has a higher chance of producing a conflicting result. With conflicts being punished heavily and non-validated answers that are deemed correct not being accounted for much, the accumulated score of a verifier returning many definitive results some of which are wrong has a high chance to score worse than a verifier returning a small number of results. Figure 7 presents updated Venn diagrams when removing all results that are in conflict with BTC.

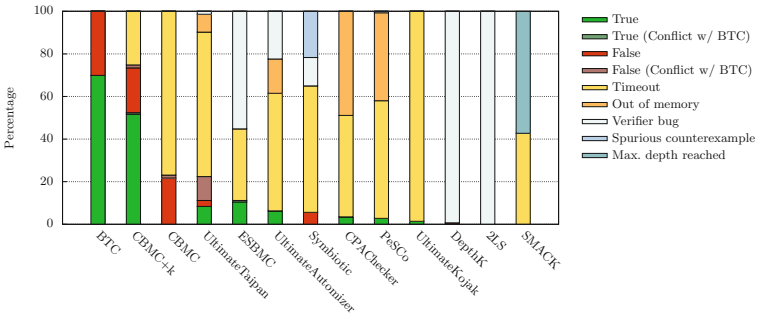

Fig. 5. The verification results for each verifier, in percent of the 143 verification tasks on which BTC returned a definite result. No witness validation results are depicted, as they were previously given in Fig. 2.

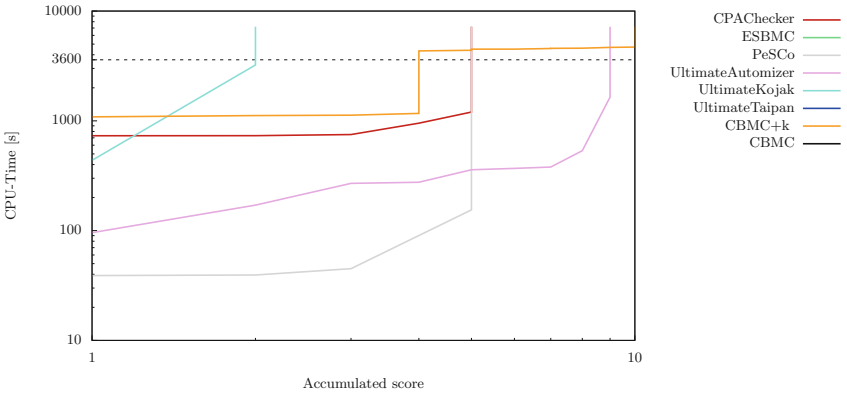

Fig. 6. The quantile plot for each verifier, assuming BTC results as ground truth. The tools 2LS, DepthK, SMACK and Symbiotic have been omitted as they do not reach a score other than zero.

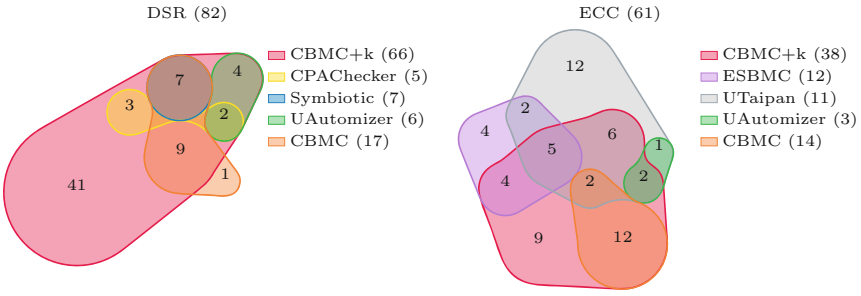

Fig. 7. Venn diagrams indicating the requirement coverage (i.e., a definite result was issued) by the top-five verifiers for case study DSR (left) and ECC (right), assuming BTC results as ground truth.

We did a careful comparison of the verification results of all verifiers. Our findings are summarized in Table 4. For the ECC case study, the verifiers gave contradicting answers for 18 requirements, i.e., about 24% of all requirements. UltimateTaipan finds violations in 16 cases, while no verifier confirms these refutations. There were no conflicts between the open-source verifiers for DSR. Three conflicts were however encountered with BTC EmbeddedValidator. In two cases, CBMC (and CBMC+k) found a counterexample at depth two, conflicting a bounded proof of BTC EmbeddedValidator of depth 10. As these requirements involve equality of floating-point numbers, there seems to be a subtle issue behind this. This can be related to different intermediate floating-point precisions being used (e.g. 64 or 80 bits) and allows for multiple different, albeit correct, conflicting results. No witnesses could be validated for any conflicting requirement, meaning that we do not have a correct measure of identifying correct answers. Because of the high complexity of the involved C code, we refrained from manual analysis. While we strongly believe in the importance of witness generation and -verification, especially in industrial applications, we want to point out that in this case, the exact results are of reduced interest—we rather want to convey the overall big picture of how well the selected open source model checkers are optimized towards real-world industrial applications.

Table 4. The contradicting results observed in DSR and ECC, respectively.

Case study	True	False	Count
DSR	CBMC+k	BTC	1
	BTC	CBMC, CBMC+k	2
			$\sum = 3$
ECC	BTC	UltimateTaipan	7
	BTC, CBMC+k	UltimateTaipan	4
	BTC, ESBMC, CBMC+k	UltimateTaipan	4
	BTC, ESBMC	UltimateTaipan	1
	ESBMC, CBMC+k	DepthK	1
	ESBMC	BTC, UltimateTaipan	1
			$\sum = 18$

5 Encountered Issues

During the course of this work we identified issues and bugs in most of the verifiers. In case we were able to identify a minimal working example, we reported bugs to the developers as noted in the footnotes below. We give a brief description of the occurring issues. Issues encountered with earlier versions of BTC EmbeddedValidator have been described in [3].

CBMC 5.11. We encountered a bug that presented itself on the code outputted by Frama-C [23], which led CBMC to report *spurious counterexamples*. In version 5.11, CBMC did not handle variables that are local to a switch block correctly and always assumed a non-deterministic value for them[6]. This bug has been fixed in subsequent releases. Additionally, when employing CBMC 5.9 or larger for CBMC+k we noticed *a drop in performance for the inductive steps* compared to version 5.8, sometimes resulting in resource exhaustion for the 5.11 version. This behavior was not emerging in the base cases, i.e., it most likely corresponds to the introduced non-deterministic state spaces, although we were not able to identify a specific cause. Lastly, CBMC outputs *witnesses that do not adhere to the format specification*[7].

ESBMC 6.0.0. On DSR, we observed *a verifier bug on 97.1%* of the verification tasks. Here, ESBMC seems to specify a faulty input for its default SMT solver, Boolector. Specifically, it appears to create if-else branching conditions of different sorts. This problem could be avoided e.g. by using Z3.

2LS 0.7.0. We identified a simple program on which 2LS delivers *false negatives*, consisting of two nested loops and a `__VERIFIER_error()` statement after the inner loop. 2LS reports such a program as safe with its k-induction setting[8]. Apart from this, 2LS did not execute on any verification task. This seems to be due to *a bug in a bit-vector map implementation*, where a size assertion fails.

CPAChecker 1.8.0. C `typedef`s were not resolved correctly[9]. This bug initially prevented the tool from running on the case studies completely, although *it was quickly fixed by the tool developers*. Furthermore, we found that switch-local variables, similar to CBMC, are not represented internally at all, and thus ignored[10]. As we tried to run CPAChecker with Z3, we were deterred by a bug in the Z3-abstraction of JavaSMT[11].

DepthK 3.1. Due to the bug exhibited by ESBMC (see above), DepthK did not execute on most of the verification tasks. Here, it creates ESBMC instances which immediately fail until DepthK reaches the time out.

SMACK 1.9.3. SMACK did not return a single definite answer, most likely due to the *default loop bound of one*.

Symbiotic 6.0.3. For both case studies, there are some properties for which KLEE prints that it is silently concretizing an expression to value 0 due to floating points, which leads to Symbiotic failing the verification. Additionally, KLEE

[6] https://github.com/diffblue/cbmc/issues/3283.
[7] https://github.com/diffblue/cbmc/issues/4418.
[8] https://github.com/diffblue/2ls/issues/123.
[9] https://groups.google.com/forum/#!topic/cpachecker-users/wTqHOedBOb0.
[10] https://groups.google.com/forum/#!topic/cpachecker-users/_bH55x_INOw.
[11] https://groups.google.com/forum/#!topic/cpachecker-users/6wv6fgwHnk4.

extracted some spurious counterexamples that it could not replay. Symbiotic stops the execution thereafter.

UltimateAutomizer 91b1670e. We observed two verification runs where UltimateAutomizer is unable to convert an assertion to an internal function representation. There are 40 ECC verification tasks leading to erroneous behavior. In 38 cases the usage of an unknown `enum` constant leads to program abortion. The remaining two instances are identical to the described bug on DSR.

UltimateTaipan 91b1670e. On DSR and ECC, the same two conversion error instances as for UltimateAutomizer apply.

6 Epilogue

This paper reported on applying 12 software model checkers to two embedded C code case studies from the automotive domain. Although this is a rather limited set of case studies, our findings give some observations that we hope to be insightful for the software verification community. From the fact that the open-source verifiers cover in the best up to 20% of all requirements—about 99% of them being invariants—makes clear that *there seems to be a serious gap between the needs of automotive code verification and open-source software model checker capabilities*. The specific characteristics of the two case studies (many floating-points, pointer dereferencing, bitwise operations etc.) are certainly a decisive factor in this respect. Additionally, the structure of an infinite outer loop (forever processing inputs) with nested finite loops seems to require an tailored k-induction to properly capture behavior, which we believe explains part of the success of CBMC+k and BTC. While both tools are heavily tailored towards special use-cases and are unsuitable for more general programs, we firmly believe these optimizations are worth pursuing and integrating into mainstream open-source verifiers. Admittedly, the fact that our benchmarks are not publicly available is a weak point. More studies like the one in this paper are needed. To that end, *the software model checking community and industrial partners covering various application domains should take up an orchestrated effort to set up a substantial set of industrial benchmarks*. The only way to meet the needs in industry is to be able to apply software model checkers on real industrial software of different domains. Finally, the results of our study (particularly, the score of CBMC+k relative to BTC) suggest *to revisit the scoring scheme of verification competitions such as SV-COMP*. In particular, the punishment of wrong verification results is too severe; it is currently measured in absolute terms (the number of wrong answers), whereas a relative judgment (what is the percentage of wrong answers that a verifier obtained) seems to be more fair.

Acknowledgment. We thank BTC Embedded Systems AG, in particular Tino Teige and Markus Gros, for their support and helpful advice. We are grateful to Md Tawhid Bin Waez and Thomas Rambow (both from Ford Motor Company) for their support on the case studies in an earlier phase and for fruitful discussions on formal verification

and Simulink. We thank Dirk Beyer for very useful feedback on an earlier version of the paper.

References

1. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reason. **60**(3), 299–335 (2018)
2. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 133–155. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_9
3. Berger, P., Katoen, J.-P., Ábrahám, E., Waez, M.T.B., Rambow, T.: Verifying auto-generated C code from simulink. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 312–328. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_18
4. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
5. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
6. 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: 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 888–891. ACM Press (2018)
7. Schrammel, P., Kroening, D.: 2LS for program analysis. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 905–907. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_56
8. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_51
9. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: FMCAD, pp. 189–197. IEEE (2010)
10. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Cortellessa, V., Varró, D. (eds.) FASE 2013. LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37057-1_11
11. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 622–640. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_42
12. Richter, C., Wehrheim, H.: PeSCo: predicting sequential combinations of verifiers. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 229–233. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_19
13. Rocha, W., Rocha, H., Ismail, H., Cordeiro, L., Fischer, B.: DepthK: a k-induction verifier based on invariant inference for C programs. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 360–364. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_23

14. Rakamarić, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 106–113. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_7
15. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
16. Chalupa, M., Vitovská, M., Jonáš, M., Slaby, J., Strejček, J.: Symbiotic 4: beyond reachability. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 385–389. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_28
17. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_2
18. Ermis, E., Nutz, A., Dietsch, D., Hoenicke, J., Podelski, A.: Ultimate kojak. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 421–423. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_36
19. Greitschus, M., et al.: Ultimate Taipan: trace abstraction and abstract interpretation. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 399–403. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_31
20. Gadelha, M.Y.R., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. STTT **19**(1), 97–114 (2017)
21. Teige, T., Bienmüller, T., Holberg, H.J.: Universal pattern: formalization, testing, coverage, verification, and test case generation for safety-critical requirements. In: MBMV, Albert-Ludwigs-Universität Freiburg (2016). P. 6–9
22. Berger, P., Nellen, J., Katoen, J.-P., Ábrahám, E., Waez, M.T.B., Rambow, T.: Multiple analyses, requirements once. In: Larsen, K.G., Willemse, T. (eds.) FMICS 2019. LNCS, vol. 11687, pp. 59–75. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-27008-7_4
23. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16