Preprint from Proceedings of GI/ITG/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", March 2021

Benchmarking SMT Solvers on Automotive Code

Lukas Mentel[1], Karsten Scheibler[1], Felix Winterer[2], Bernd Becker[2], Tino Teige[1]
[1] BTC Embedded Systems AG, Oldenburg, Germany, {lukas.mentel,karsten.scheibler,tino.teige}@btc-es.de
[2] University of Freiburg, Freiburg, Germany, {winteref,becker}@informatik.uni-freiburg.de

Abstract

Using embedded systems in safety-critical environments requires a rigorous testing of the components these systems are composed of. For example, the software running on such a system has to be evaluated regarding its code coverage – in particular, unreachable code fragments have to be avoided according to the ISO 26262 standard. Software model checking allows to detect such dead code automatically. While the recent case study [1] compares several academic software model checkers with the commercial test and verification tool BTC Embedded*Platform*® (BTC EP), we want to focus on a lower level – i.e. the back-end solvers within BTC EP. Therefore, we evaluate the performance of off-the-shelf SMT solvers supporting the theory of floating-point as well as the theory of bitvectors on floating-point dominated benchmark instances originating from the automotive domain. Furthermore, we compare these off-the-shelf SMT solvers with the back-end solvers used by BTC EP.

1 Introduction

In safety-critical system development like in the automotive and aviation industry there is the imperative for high-quality embedded production code. This is also reflected by several existing standards, e.g. the ISO 26262 standard [2] for the automotive domain which objects to *dead code* – i.e. unreachable code fragments. A classic example of dead code is defensive programming like catching potentially non-existing null pointers. In such cases, a clear and well comprehensible justification for the existence of dead code is required to assure the code quality.

In order to prove the absence of dead code, the ISO 26262 recommends several code coverage metrics like statement coverage, condition coverage, decision coverage, or modified condition/decision coverage (MC/DC) [3]. Such code coverage metrics implement a measure to which degree the source code was executed during the test stage by means of test cases. If the coverage value is 100% then no dead code is left according to the applied coverage metric. If, however, the coverage value is below 100% then there is a potential risk of dead code which requires a further analysis.

In order to automatically detect dead code, software model checking can be applied. One distinctive feature of the commercial test and verification tool suite BTC Embedded*Platform*®[1] (BTC EP) is the fully automatic detection of dead code [4] by employing several model checking tools like CBMC [5, 6] and iSAT3 [7]. On account of these model checkers, BTC EP also has a very strong support for the automatic analysis of *floating-point* dominated production code, which becomes more and more apparent in industrial safety-critical software [8].

According to [9] there are four "schools of thought" when it comes to SMT-based software verification: bounded model checking, *k*-induction, predicate abstraction and lazy abstraction with interpolants. In the recently published case study [1] several academic state-of-the-art software model checkers were evaluated and compared to BTC EP. The considered C code was automatically generated from two Simulink open-loop controller models provided by Ford Motor Company. The diverse features of both models (decision logic, floating-point arithmetic, rate limiters and state-flow systems) and in particular the floating-point dominated C code are a challenging task for software model checkers. The authors of [1] address two questions in their case study: (1) how do academic model checkers perform on automotive code, and (2) how do these tools compare to commercial tools that are tailored to such code. This is the essence of the results: on the 179 software requirements the academic model checkers were only able to cover at most 20% while BTC EP succeeded on 80%. Thus, there is a considerable gap between the performance of the academic model checkers and BTC EP.

While in [1] the comparison was performed on the level of software model checkers, we focus on a lower level – the back-end solvers. This means, in this paper we compare the two solvers CBMC and iSAT3 (being integrated in BTC EP) with other SMT solvers supporting the theory of bitvectors as well as the theory of floating-point. It should be noted that both theories are required – as C code with floating-point arithmetic usually contains integer arithmetic as well. In our evaluation we consider models being provided by BTC EP users from the industrial automotive domain. The considered models are not identical to the models of [1] but have similar characteristics. These models are translated via TargetLink to C code which is then passed to BTC EP in order to apply automatic test case generation. The resulting instances are the basis of our evaluation. In each instance the reachability of a code fragment has to be determined. If a code fragment is reachable, the solver has to provide a test case. Furthermore, the solver has to be able to prove that a code fragment is unreachable and is thus dead code. Hence, Bounded Model Checking is not sufficent in this context. But the control-

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

flow structure in the instances allows a direct application of k-induction – this comes due to the special characteristics of the C code which encodes an open-loop controller. With our evaluation we want to answer similar questions as [1]:

1. How do off-the-shelf SMT solvers perform on instances originating from automotive code?
2. How do these solvers compare to the back-end solvers used in the commercial tool BTC EP – in particular when considering the default time limit of 60 seconds of BTC EP?

The structure of the paper is as follows. While Section 2 introduces the considered solvers, Section 3 describes the benchmark instances being solved by all solvers. The experimental results are evaluated in Section 4. Finally, Section 5 concludes the paper.

2 Solvers

Before introducing the solvers, we give a brief overview of the techniques used by the solvers – namely BMC, k-induction and Craig Interpolation.

Informally, when perfoming model checking the system to be verified is abstracted into a set of states and a relation which represents allowed transitions from one state to another. Thus, this transition relation restricts how the state of the system evolves over time. Let I denote the set of initial states of the system and let T denote the transition relation. Usually, it is checked whether a certain set of states (denoted by P) is never left – or in other words: it is checked whether it is possible to reach states outside of P, i.e. states in $\neg P$.

Representing I, T and $\neg P$ as SMT formulas allows the application of Bounded Model Checking (BMC) [10] in order to determine whether it is possible to reach states in $\neg P$. This is achieved by solving a sequence of formulas – one formula for each transition step $0,1,\ldots,k$. The resulting formulas are composed of instances of I, T and $\neg P$. The subscript indicates in which time frame the formula is instantiated.

$$\begin{aligned}
\mathrm{BMC}_0 :&\quad I_0 \wedge \neg P_0 \\
\mathrm{BMC}_1 :&\quad I_0 \wedge T_{0,1} \wedge \neg P_1 \\
\mathrm{BMC}_2 :&\quad I_0 \wedge T_{0,1} \wedge T_{1,2} \wedge \neg P_2 \\
&\quad \ldots \\
\mathrm{BMC}_k :&\quad I_0 \wedge \left(\bigwedge_{i=0}^{k-1} T_{i,i+1} \right) \wedge \neg P_k
\end{aligned}$$

The formulas are solved step-by-step as long as they are unsatisfiable. If a BMC_i formula becomes satisfiable, it is proven that states contained in $\neg P$ are reachable in i transition steps. On the other hand, if all formulas up to BMC_k are unsatisfiable, it is only proven that the states in $\neg P$ are unreachable in up to k transition steps – but it would be more desirable to know that this is always the case, i.e. not bounded by k but unbounded for all transition steps. Obtaining an unbounded result can be achieved by exploiting the knowledge proven by the BMC formulas. If all formulas up to BMC_{k-1} are unsatisfiable, it is known that the reachable states in up to $k-1$ transition steps are within P. Hence, this knowledge can be incorporated into BMC_k:

$$\mathrm{BMC}_k : \quad I_0 \wedge \left(\bigwedge_{i=0}^{k-1} (P_i \wedge T_{i,i+1}) \right) \wedge \neg P_k$$

Let the suffix of BMC_k be denoted by IND_k:

$$\mathrm{IND}_k : \quad \left(\bigwedge_{i=0}^{k-1} (P_i \wedge T_{i,i+1}) \right) \wedge \neg P_k$$

Thus, if IND_k is unsatisfiable, it is proven that all BMC_i with $i > k$ will be unsatisfiable as well – as they contain a similar suffix when neglecting variable renaming due to different time frame instantiations. Basically, solving an alternating sequence of BMC and IND formulas is the idea of the k-induction approach presented in [11].

Craig Interpolation [12] is another approach in order to obtain unbounded results. This approach also exploits knowledge gained during the solving process of a BMC formula – but in quite a different way. If a BMC formula is unsatisfiable, the solver has internally built a proof tree. Based on this proof tree an interpolant is created which represents an overapproximation of the reachable states. Afterwards, BMC-like formulas are solved – i.e. I is replaced with an interpolant and further interpolants are created. Additionally, it is checked whether the created interpolants reach a fixed-point – in such a case it is proven that all states in $\neg P$ are unreachable. On the other hand, if a BMC-like instance becomes satisfiable it is required to build the according BMC_i instance and solve it from scratch in order to check whether there is indeed a solution.

While k-induction does not require special support from the used solver, Craig Interpolation requires access to the proof tree in order to create interpolants. Therefore, in our evaluation most of the solvers rely on k-induction – only iSAT3 uses Craig Interpolation.

With IC3 [13] and its extended variant PDR [14] there is a third approach for unbounded results. Similar to Craig Interpolation, IC3 requires solver support in order to perform an efficient generalization. Although IC3 is included in a recent prototype of iSAT3 [15, 16], we do not consider this prototype within this paper as we focus on the techniques that are available in the current release version of iSAT3 being included in BTC EP. Furthermore, we do not consider the SMT solvers XSat [17] and goSAT [18]. Both solvers support the theory of floating-point – but cannot handle the theory of bitvectors. Thus, they are incapable of solving the instances considered in this paper. Additionally, both solvers are incomplete – i.e. they cannot prove that a formula is unsatisfiable and are therefore unable to detect dead code. We evalute the following solvers:

- Z3 [19] in version 4.8.9[2].
- CVC4 [20] in version 1.8[3].
- MathSAT5 [21] in version 5.6.5[4].
- Bitwuzla [22] in version 20201208[5]. Bitwuzla is the successor of Boolector [23].

[2]Taken from https://github.com/Z3Prover/z3/releases/
[3]Taken from https://github.com/CVC4/CVC4/releases/
[4]Taken from https://mathsat.fbk.eu/download.html
[5]Provided by the Bitwuzla authors via Email

- CBMC [5, 6] in version 5.12.4[6]. We use CBMC within our wrapper tool EP-CBMC which also implements *k*-induction.
- iSAT3 [7] in version 0.08.1-20201103[7].

```
; SMT2: Z3, CVC4, MathSAT5, Bitwuzla

(set-logic QF_BVFP)
(set-option :produce-models true)
(declare-fun x () Float64)
(declare-fun y () Float64)
(declare-fun z () Float64)
(define-fun const_1 () Float64
  (fp #b0 #b01111111111 #x0000000000000))
(assert (fp.eq (fp.div RNE (fp.div RNE x y) z) const_1))
(check-sat)
(get-model)
(exit)
```

```
/* C: CBMC */

double nondet_double(void);

int main(void) {
  double x = nondet_double();
  double y = nondet_double();
  double z = nondet_double();
  int result = 0;

  if ((x / y) / z == 1.0) result = 1;
  __CPROVER_assert((result == 0), "result == 0");
  return (0);
}
```

```
-- extended HYS: iSAT3

DECL
-- NaN is always allowed and not part of the interval
cl_double [cl_double_neginf,cl_double_posinf] x, y, z;

EXPR
define const_1 = cl_double_constant(1);
define div_1 = cl_double_div(x, y);
define div_2 = cl_double_div(div_1, z);
cl_double_equal(div_2, const_1);
```

Solver	Runtime in seconds	Memory in MB
CVC4	7051.36	12320
Z3	214.53	605
MathSAT5	2.13	149
Bitwuzla	0.33	65
CBMC	0.19	30
iSAT3	0.01	3

Figure 1 Small example benchmark with two floating-point divisions in three different encodings: SMT2, C and HYS. The runtime and memory needed by each solver is shown as well.

The solvers Z3, CVC4, MathSAT5, Bitwuzla natively support the SMT2 format while CBMC requires a C program as input – iSAT3 expects its input to be in HYS format. This has historical reasons as iSAT3 and its predecessors HySAT [24] and iSAT [25, 26, 27] were originally developed for the verification of hybrid systems. Although all three solvers support linear and non-linear arithmetic as well as transcendental functions, they expect that every variable is bounded – which is a reasonable assumption in the context of hybrid systems. With this restriction the support for the SMT2 format did not made sense as SMT2 is not aware of explicit variable bounds. After adding the support for accurate floating-point reasoning in iSAT3 [28] there was no pressing need to add support for the SMT2 format – in particular, as the HYS input language provides cast operations between floating-point and bitvector types which are not directly available in SMT2[8].

The solvers Z3, CVC4, MathSAT5, Bitwuzla and CBMC use *bit-blasting* to translate the given SMT formula into a SAT formula. The translation of CVC4 and Bitwuzla can be seen as a two-stage process, because both solvers natively support the theory of bitvectors and use SymFPU [29] to encode floating-point operations at the level of bitvectors. Furthermore, according to [30] and [22] both solvers use CaDiCaL [31] as back-end SAT solver.

We use CBMC with the command line option `--refine`. With this option CBMC starts with a coarser encoding of the floating-point values and refines the encoding in case of spurious solutions. In our experience this helps to improve the overall performance.

Besides bit-blasting there exist interval based approaches to solve SMT formulas with floating-point arithmetic. In particular, iSAT3 [28, 7] builds on this technique. Furthermore, with the approach of [32] there is an interval-based floating-point reasoning available for MathSAT5 – but to the best of our knowledge this implementation is not enabled by default. We refer the reader to [28] for a comparison with iSAT3.

Usually, interval-based techniques have advantages for operations like addition, subtraction, multiplication and division in comparison to bit-blasting. Figure 1 shows a small example with two floating-point divisions encoded in SMT2, C and HYS to demonstrate this. In particular the bit-blasting based solver CVC4 is unable to solve this instance within one hour while iSAT3 solves it immediately. Bitwuzla and CBMC perform quite well – probably due to further optimizations, e.g. stochastic local search [22]. On the other hand, bitwise operations are usually handled more efficiently by solvers based on bit-blasting compared to interval-based solvers. Thus, depending on the characteristics of the instance to be solved one technique might outperform the other.

As CBMC has no built-in support for *k*-induction, the tool chain within BTC EP prepares different input files for CBMC in order to perform BMC and IND checks. Thus, multiple CBMC calls are required for *k*-induction – for a better distinguishability we call the resulting wrapper EP-CBMC. In order to reduce the number of calls, only IND_2 is considered in the default settings of EP-CBMC.

Regarding the SMT2 solvers Z3, CVC4, MathSAT5 and Bitwuzla the instances are prepared differently. While CBMC and iSAT3 create and solve the BMC_i formulas internally, the SMT2 format does not provide a direct support for BMC. On the other hand, it is possible to interact with an SMT2 solver by redirecting the standard input and output channels. Furthermore, the `push` and `pop` statements within the SMT2 language allow to add and remove formula parts. Thus, it is possible to solve an alternating sequence of BMC and IND formulas with an SMT2 solver.

[6]Taken from https://github.com/diffblue/cbmc/releases/

[7]Not publicly available, included in BTC EP 2.8.

[8]The cast operations in iSAT3 always have defined behavior.

		Floating-point variables													
		0	1	2-4	5-9	10-14	15-19	20-34	35-49	50-74	75-99	100-149	150-199	200-249	Σ
Integer variables	0	362		243	172	107	32	20							936
	1			5	22	66		136	16						245
	2-4	392		342	70	103		136	12						1055
	5-9	128	7	566		172		8	2						883
	10-14	150		11											161
	15-19	40		30	54	2									126
	20-34	83		26	25	15									149
	35-49			40										78	118
	50-74			22	29										51
	75-99							775	452						1227
	100-149								1282	1562					2844
	150-199				983										983
	Σ	1155	7	1285	1355	465	32	1075	1764	1562	0	0	0	78	8778

Table 1 An overview of how many benchmark instances have a certain number of integer and floating-point variables. For a compact representation the variable numbers are bucketized, i.e. each line relates to a range of integer variables while each column relates to a range of floating-point variables. For example, there are 78 instances containing between 200 and 249 floating-point variables as well as between 35 and 49 integer variables. Furthermore, there are 362 instances containing only Boolean variables – showing the effect of the cone-of-influence reduction, i.e. not every instance derived from floating-point dominated C code is required to contain floating-point or integer variables. Auxiliary variables for subexpressions are not counted here. Furthermore, the numbers relate to one step – for a BMC instance with k steps the number of variables has to be multiplied by k.

3 Benchmarks

The 8778 benchmark instances considered in this paper encode code coverage *goals* – each instance corresponds to one goal, i.e. a code fragment whose reachability has to be determined. In the following we describe the goal derivation in more detail. Additionally, we provide some statistics regarding the resulting benchmark instances – e.g. regarding the number of floating-point variables or the used operations.

Starting from a set of Simulink models provided by BTC EP users from the industrial automotive domain, TargetLink was used to automatically generate production C code for these models. Thus, the obtained C code obeys a specific structure. It can be understood as a reactive program with an unbounded feedback loop divided into three parts: (1) all input values for the current *step* are received, (2) the actual function is executed, and (3) the computed output values are transmitted. Afterwards, the program waits for the next loop iteration.

In order to analyze the coverage of the C code, the code is automatically instrumented to collect coverage information [4]. In a simple form of such an instrumentation each basic block in the C code is accompanied with an auxiliary Boolean variable – also called a goal. This auxiliary variable is initialized with false and is set to true if the basic block is entered. When considering more advanced coverage metrics like modified condition/decision coverage (MC/DC) [3], the instrumentation decomposes an `if` condition into its Boolean components in order to collect which truth value combinations of the Boolean components were responsible for entering a basic block.

In order to automatically generate test cases, BTC EP translates the C code internally into an intermediate language called SMI. At SMI level further transformations are performed – in particular, inlining of function calls and flattening of complex data structures. As each goal is processed individually, it is beneficial to apply a cone-of-influence reduction in order to remove all code fragments being irrelevant for the goal under consideration. Furthermore, depending on the selected back-end solver additional rewritings are performed on SMI level: (1) every loop is unrolled, and (2) the resulting code is transformed into a static single assignment (SSA) form. In the context of this paper, both techniques are always applied.

It is easily possible to translate the SSA form into an SMT formula. Basically, the resulting SMT formula represents one iteration (i.e. one step) of the unbounded feedback loop of the reactive program which encodes the originating Simulink model. Hence, in order to analyze up to k iterations, BMC can be applied (cf. Section 2). Thus, if an instance becomes satisfiable in BMC depth k the goal is reachable in k steps. Furthermore, the obtained satisfying assignment corresponds to a test case of the production C code – certifying the coverage of the goal under consideration. On the other hand, it is also very important to know whether a goal is unreachable as this allows to adjust the coverage information – which is very important in practice in order to conform to standards like ISO 26262 [2]. As explained in Section 2, the evaluated solvers use k-

	1-9	10-99	100-999	1000-9999	10000+
Nr. subexpr.	234	1269	1403	3734	2138

Table 2 The number of subexpressions used in the benchmark instances. Usually, a subexpression contains between one and three operations. As listed, there are 2138 instances with more than 10000 subexpressions. The numbers relate to one step – for a BMC instance with *k* steps the number of subexpressions has to be multiplied by *k*.

induction or Craig Interpolation to detect unreachable code fragments.

The resulting 8778 benchmark instances contain a mix of Boolean variables, floating-point variables and fixed bit-width integer variables. Table 1 gives an overview of the number of floating-point and integer variables contained in the instances. The shown numbers relate to one step, i.e. if a BMC instance with 3 steps is considered the numbers have to be tripled. Furthermore, these variables either represent input values or local state variables – auxiliary variables representing the value of a subexpression are not included. Table 2 shows the size of the instances when considering the contained subexpressions. The CBMC encoding requires the declaration of an additional variable for each subexpression while the SMT2 encoding uses `define-fun` – iSAT3 uses a similar definition in its HYS input language.

Table 3 shows the usage of different operations in the benchmark instances. It should be noted that while C allows different operand types for an operation (e.g. adding an integer and a floating-point variable), this is not allowed in SMI – i.e. explicit casts are required if the operands of an operation have different types. The following operation kinds are listed in Table 3:

- Casts operations: (1) between different floating-point formats, i.e. `float` and `double`, (2) between integer and floating-point formats, and (3) between different integer bit-widths.
- Comparison operations regarding floating-point and integer data types, respectively.
- Addition and subtraction operations involving floating-point and integer types, respectively.
- Multiplication and division operations involving floating-point and integer types, respectively.
- Bitwise integer operations.

It can be observed that the benchmark instances contain fewer bitwise operations than arithmetic operations – in particular floating-point arithmetic is used very frequently within the instances.

	1-9	10-99	100-999	1000-9999	10000+
Casts fp $\leftrightarrow$ fp	2313	4158			
Casts int $\leftrightarrow$ fp	784	1119			
Casts int $\leftrightarrow$ int	4066	2771	287		
Comparisons fp	242	3227	4159		
Add / sub fp	1488	3819	904		
Mul / div fp	468	2578	452		
Comparisons int		1526	1513	4780	959
Add / sub int	2651	2619	1843		
Mul / div int	1542				
Bitwise int	1095				

Table 3 An overview of the operations used in the benchmark instances. For each kind of operation (e.g. integer addition and subtraction) the occurrence is divided into buckets. For example, there are 959 instances with more than 10000 integer comparison operations. The numbers relate to one step – for a BMC instance with *k* steps the number of operations has to be multiplied by *k*.

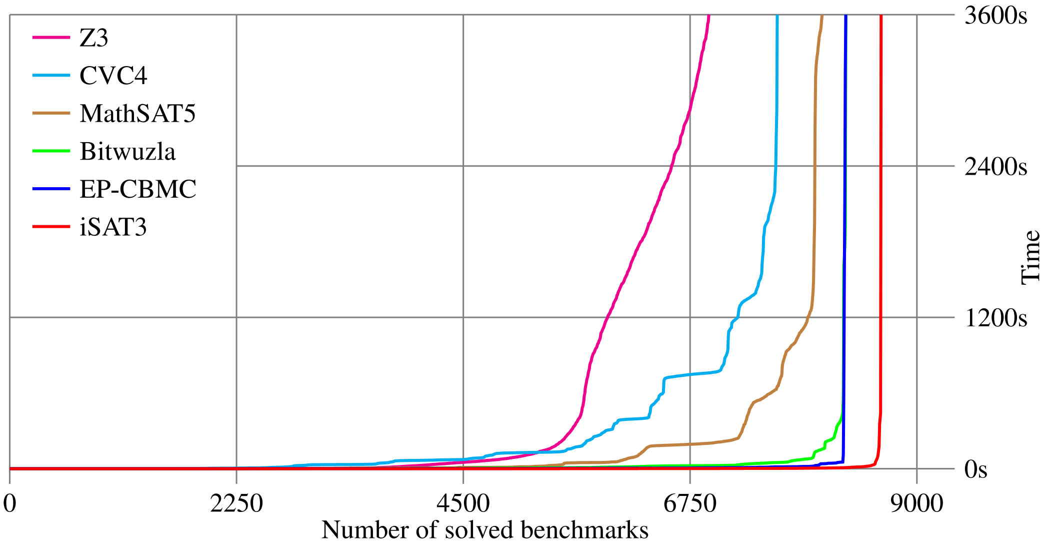
4 Experimental Results

We used Z3, CVC4, MathSAT5 and Bitwuzla without any additional command line options. CBMC and iSAT3 were used with their BTC EP default settings. The experiments for Z3, CVC4, MathSAT5, Bitwuzla and iSAT3 were performed on a Linux cluster with Ubuntu 20.04. Each cluster node had 64 GB RAM and two 8-core CPUs running at 2.6 GHz. Per benchmark instance we applied a time limit of 3600 seconds and a memory limit of 8 GB.

For technical reasons EP-CBMC could not be executed in this environment – instead we executed it with the same limits on a computer with a similar CPU also running at 2.6 GHz. Furthermore, the runtime of EP-CBMC also includes the time to translate SMI to C while the runtime of all other solvers does not include this overhead. In most cases the translation time is negligable in comparison to the solving time. Nonetheless, the effect is noticable for instances with very low solving times, i.e. in the second diagram of Figure 2 the blue CBMC curve is slightly above the curves of all other solvers for the first 2000 instances – this is due to the translation overhead. Hence, the number of solved instances of EP-CBMC might increase slightly in a perfect comparison.

In practice, we observed that in most cases it suffices to perform 2-induction – i.e. solving IND_2. Applying *k*-induction for every transition step just slows down the solving process without increasing the number of detected dead code instances noticably[9]. Therefore, EP-CBMC applies 2-induction per default. Additionally, this allows to em-

[9]While EP-CBMC finds 628 dead code instances with 2-induction, this number increases only slightly to 632 when applying *k*-induction in every transition step.



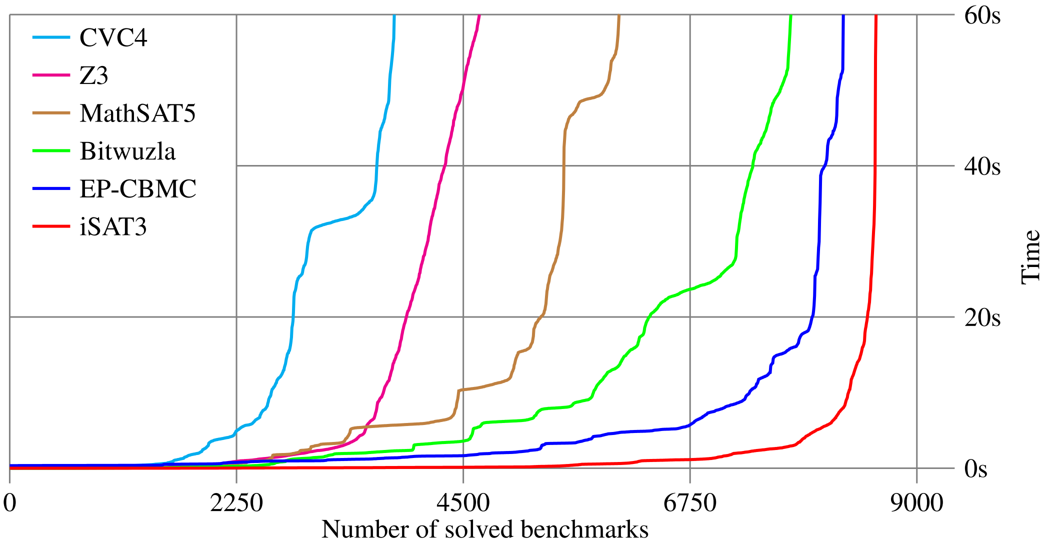


Figure 2 Comparison between Z3, CVC4, MathSAT5, Bitwuzla, CBMC and iSAT3 on a set of 8778 benchmark instances originating from automotive code. While the first diagram is regarding a timeout of 3600 seconds, the second diagram considers only instances solved within 60 seconds – as in practice this is usually the time limit per instance for a solver within BTC EP.

ploy the incremental BMC support of CBMC [6] for BMC$_i$ with $i \geq 2$. In contrast, iSAT3 applies Craig Interpolation in every transition step. Furthermore, in order to certify a solution, iSAT3 needs to solve the BMC$_i$ instance from scratch if the according BMC-like formula (which contains an interpolant instead of I) becomes satisfiable in transition step i (cf. Section 2).

All four SMT2 solvers get an input file per instance. Each file employs incremental solving by using `push` and `pop` statements in order to solve a sequence of BMC formulas and one IND$_2$ formula – similar to EP-CBMC we use 2-induction.

4.1 Number of Solved Instances

Figure 2 gives an overview of the number of solved instances for all six solvers. While the first diagram considers a timeout of 3600 seconds, the second diagram zooms into the number of instances being solved within 60 seconds – as in practice this is the time limit solvers have to operate with when used within BTC EP.

With a timeout of 3600 seconds Z3 solves the fewest number of instances followed by CVC4 and MathSAT5. Bitwuzla and EP-CBMC seem to perform equally well – except a small difference at the bottom of the diagram. The highest number of instances is solved by iSAT3.

The picture changes when considering a timeout of 60 seconds. This timeout is chosen deliberately as this is the default time limit per instance within BTC EP. Compared to the timeout of 3600 seconds there are two changes: (1) CVC4 solves the fewest instances followed by Z3 and MathSAT5, and (2) there is a noticable gap between Bitwuzla and EP-CBMC, i.e. the small difference between both solvers at the bottom of the first diagram now becomes relevant as EP-CBMC solves considerably more instances than Bitwuzla. As before, iSAT3 solves the highest number of instances.

Table 4 gives a more detailed view to the solved instances. It can be observed that EP-CBMC and iSAT3 perform equally well regarding the number of test cases (TC) being found – in absolute numbers EP-CBMC performs slightly better. This can be attributed to the fact that EP-CBMC uses incremental BMC starting from transition step 2 while iSAT3 with Craig Interpolation has to solve an additional complete BMC formula from scratch in order to certify the solution. If Bitwuzla is given enough time, it finds two TC instances more than EP-CBMC. Furthermore, with a timeout of 3600 seconds Bitwuzla and EP-CBMC prove 628 instances to be dead code (DC) – with the 60 seconds timeout Bitwuzla loses 15 instances while EP-CBMC stays nearly constant with 627 instances. Due to Craig Interpolation iSAT3 outperforms all other solvers concerning the number of DC instances – regardless of the timeout. Additionally, the second diagram of Figure 2 reveals that with even lower timeouts (e.g. 10 seconds) iSAT3 finds the highest number of TC instances – this is not surprising as the set of instances contain few bitwise operations and many arithmetic operations which is advantageous for iSAT3 with its interval-based approach.

	Timeout 60 sec.			Timeout 3600 sec.		
Solver	TC	DC	T/M	TC	DC	T/M
Z3	4123	537	4118	6359	577	1842
CVC4	3431	384	4963	7074	541	1163
MathSAT5	5624	423	2731	7434	626	718
Bitwuzla	7135	613	1030	7666	628	484
EP-CBMC	7642	627	509	7664	628	486
iSAT3	7624	970	184	7662	983	133

Table 4 The number of found test cases (TC) and instances with dead code (DC) regarding a timeout of 60 seconds and 3600 seconds, respectively. Furthermore, the number of unsolved instances due to timeout or memout (T/M) is shown as well.

Regarding the number of TC and DC instances Bitwuzla outperforms the remaining three SMT2 solvers remarkably. When considering only the solved TC instances, then MathSAT5 is closest to Bitwuzla. Regarding solved DC instances, it depends on the timeout whether Z3 or MathSAT5 perform better. In general, it seems Z3 performs quite well on DC instances – in particular compared to CVC4. Thus, when considering TC and DC instances separately, Z3 outperforms CVC4 clearly for DC instances while CVC4 solves more TC instances when given enough time.

The BMC problems of the TC instances become satisfiable in most cases in less than 10 transition steps – but there are also a few instances which require BMC to be performed until transition step 483 before the formula becomes satisfiable. Thus, another aspect is whether these instances are solved as well. While EP-CBMC and iSAT3 solve some of these TC instances already within 60 seconds, Bitwuzla requires 137 seconds to solve the first instance of this kind – MathSAT5 needs 177 seconds. In contrast, Z3 and CVC4 are unable to solve such instances even within 3600 seconds.

4.2 Number of Uniquely Solved Instances

Table 5 addresses another aspect to judge the results: the number of uniquely solved instances. In particular this number is of interest when considering a portfolio of multiple solvers – as adding a further solver to an existing portfolio is only beneficial if the added solver contributes uniquely solved instances. When solving floating-point instances, BTC EP relies on EP-CBMC and iSAT3. Thus, the question is whether the SMT2 solvers are able to solve instances which were not solved by EP-CBMC or iSAT3. As shown in Table 5 this is not the case. When considering EP-CBMC and iSAT3 individually, then in particular Bitwuzla has some uniquely solved instances – but there would be no benefit to add Bitwuzla to the existing portfolio containing EP-CBMC and iSAT3.

On the other hand, when considering today's multicore workstations, then it can be beneficial to run multiple solvers in parallel. Although no SMT2 solver is able to solve more instances than the combination of EP-CBMC and iSAT3 and thus the number of instances being solved within a 60 seconds timeout would stay unchanged, the overall runtime might decrease if a SMT2 solver returns a result in shorter time. The number of such instances is quite low, i.e. CVC4 is faster for 6 instances, MathSAT5 is faster for 7 instances, Z3 is faster for 8 instances and Bitwuzla is faster for 14 instances. Thus, the benefit of adding one of the SMT2 solvers to a parallel portfolio would be marginal.

4.3 Separating BMC and IND Formulas

While we think there is only limited room for improvements regarding the encoding of the operations in SMT2[10], there is optimization potential when separating the BMC and IND formulas. As mentioned earlier, each file passed to the four SMT2 solvers employs incremental solving by using `push` and `pop` statements in order to solve a sequence of BMC formulas and one IND_2 formula. Thus, in order to solve BMC and IND formulas within one instance, the initial states have to be removed. Depending on the implementation of the SMT2 solver this can deteriorate the solving performance as learned knowledge which involves the initial states might be invalidated. A similar observation was made in the original k-induction paper [11] – according to the experimental results it is beneficial to use two separate instances for BMC and IND formulas.

In order to test the effect of such a separation, we concentrate on the BMC part and use a plain incremental BMC encoding without any IND check. Thus, regarding the TC instances we optimistically assume that the IND_2 check (which will be always satisfiable) adds no runtime. With this approach we want to evaluate how the four SMT2 solvers would perform under optimal conditions.

[10]When neglecting syntax, the transition step in an SMT2, C and HYS instance is encoded in a similar way.

Solver	TC	EC uniq. TC	i3 uniq. TC	EC+i3 uniq. TC	DC	EC uniq. DC	i3 uniq. DC	EC+i3 uniq. DC
	Timeout 60 seconds							
Z3	4123	0	0	0	537	0	1	0
CVC4	3431	0	0	0	384	0	2	0
MathSAT5	5624	0	0	0	423	0	2	0
Bitwuzla	7135	1	2	0	613	1	12	0
EP-CBMC	7642	-	30	-	627	-	19	-
iSAT3	7624	12	-	-	970	362	-	-
	Timeout 3600 seconds							
Z3	6359	0	0	0	577	0	2	0
CVC4	7074	1	0	0	541	0	2	0
MathSAT5	7434	1	0	0	626	0	16	0
Bitwuzla	7666	3	12	0	628	0	18	0
EP-CBMC	7664	-	13	-	628	-	18	-
iSAT3	7662	11	-	-	983	373	-	-

Table 5 Overview of uniquely solved TC and DC instances regarding EP-CBMC (abbreviated EC), iSAT3 (abbreviated i3) and the union of EP-CBMC and iSAT3 (abbreviated EP+i3). For example, when considering a timeout of 3600 seconds EP-CBMC solves 13 TC instances which were not solved by iSAT3 while iSAT3 solves 11 TC instances being unsolved by EP-CBMC. Furthermore, it can be observed that no SMT2 solver is able to solve instances which were not solved by EP-CBMC or iSAT3 – i.e. both columns for EP+i3 contain only zeros.

Table 6 shows the results for the number of solved TC instances. We compare with the numbers of EP-CBMC and iSAT3 from Table 4 – i.e. while the SMT2 solvers solve pure BMC instances, EP-CBMC and iSAT3 still perform *k*-induction and Craig Interpolation, respectively. For an easier comparison, column 2 in Table 6 includes in parenthesis the number of solved TC instances from Table 4. It can be observed that the number of instances solved by the SMT2 solvers increases considerably in most cases with the pure BMC encoding – e.g. MathSAT5 solves more than 1000 additional TC instances within the 60 seconds timeout. Furthermore, with a timeout of 3600 seconds Bitwuzla solves 12 instances which were not solved by any other solver. On the other hand, the number of instances being solved faster by Z3, CVC4, MathSAT5 or Bitwuzla compared to EP-CBMC and iSAT3 increases only marginally and stays below 30 for each SMT2 solver.

As observed in Section 4.1 with a timeout of 3600 seconds Bitwuzla already performs equally well as EP-CBMC regarding the number of solved instances – while EP-CBMC clearly outperforms Bitwuzla when considering a 60 seconds timeout. When using two SMT2 solver instances in order to solve the BMC and IND formulas separately, it can be expected that Bitwuzla reduces the distance to EP-CBMC and iSAT3 regarding the 60 seconds timeout – but even under optimal conditions for Bitwuzla (i.e. assuming IND_2 causes no runtime) EP-CBMC and iSAT3 still solve roughly 150 TC instances more. Similarly, although MathSAT5 now solves more than 1000 additional TC instances, MathSAT5 is even under optimal conditions still roughly 900 instances behind EP-CBMC and iSAT3 – the gap from EP-CBMC and iSAT3 to CVC4 and Z3 is even larger. Thus, the overall ranking of the solvers stays unchanged.

4.4 Encountered Issues

Bitwuzla is a new solver first presented in 2020. Thus, it can be expected that there might be issues when running the solver on benchmark instances it was not tested on. During our experiments we observed that Bitwuzla terminated abnormally on some instances – this was quickly fixed by the authors of Bitwuzla.

On the other hand, we did not expect problems with the more mature solvers Z3, CVC4 and MathSAT5. Hence, we were surprised that Z3 and the Windows version of MathSAT5 returned spurious results in some cases. The issue in MathSAT5 is already fixed in version 5.6.5. Regarding Z3 the affected instances are rather large – therefore, we try to extract a smaller fragment still triggering the issue. Just submitting the original instances to a public issue tracker is not an option for us as the instances considered in this paper are confidential.

5 Conclusion

Inspired by the evaluation of [1] which benchmarked software model checkers on automotive code, we benchmarked in this paper SMT solvers on automotive code. The automotive code considered here is not identical to [1] but has similar characteristics. Our evaluation is motivated by the question whether off-the-shelf SMT solvers could supplement our current back-end solvers for floating-point instances within BTC EP – namely EP-CBMC (which is a wrapper for CBMC) and iSAT3. Thus, we compared both solvers with SMT solvers supporting the theory of bitvectors and the theory of floating-point. In order to reduce the effort regarding file formats, we concentrated on SMT solvers supporting the SMT2 format and selected Z3, CVC4, MathSAT5 and Bitwuzla.

Solver	Timeout 60 seconds				
		TC	EC uniq. TC	i3 uniq. TC	EC+i3 uniq. TC
Z3	(4123)	4794	0	0	0
CVC4	(3431)	4191	0	0	0
MathSAT5	(5624)	6716	0	0	0
Bitwuzla	(7135)	7471	2	21	0
EP-CBMC		7642	-	30	-
iSAT3		7624	12	-	-

	Timeout 3600 seconds				
Z3	(6359)	6577	0	0	0
CVC4	(7074)	7288	1	0	0
MathSAT5	(7434)	7561	1	0	0
Bitwuzla	(7666)	7659	13	12	12
EP-CBMC		7664	-	13	-
iSAT3		7662	11	-	-

Table 6 Overview of the number of solved TC instances regarding a pure incremental BMC encoding for the SMT2 solvers by optimistically assuming that the IND_2 check adds no runtime. The numbers of the uniquely solved TC instances are regarding EP-CBMC (abbreviated EC), iSAT3 (abbreviated i3) and the union of EP-CBMC and iSAT3 (abbreviated EP+i3). The numbers in parenthesis for the SMT2 solvers are taken from Table 4 to allow an easier comparison. Even under this optimal condition no SMT2 solver is able to outperform EP-CBMC or iSAT3.

The instances used in our experiments originate from automatic test case generation. Besides generating test cases, it is additionally required that the solvers are able to detect dead code. Therefore, we used k-induction for the SMT2 solvers. Our SMT2 encoding includes BMC and k-induction checks in one instance. Additionally, we tested the effect of separating BMC and k-induction formulas. Hence, when coming back to the two questions from Section 1 the answers are as follows:

1. *How do off-the-shelf SMT solvers perform on instances originating from automotive code?*

   In order to answer this question, we used a timeout of 3600 seconds – i.e. we wanted to know how many instances can be solved at all when a solver has enough time. Regarding this timeout the four tested SMT2 solvers are ranked as follows: Bitwuzla is the clear winner (8294 solved instances), followed by MathSAT5 (8060 solved instances), CVC4 (7615 solved instances) and Z3 (6936 solved instances). On the other hand, none of the SMT2 solvers was able to outperform EP-CBMC (8292 solved instances) or iSAT3 (8645 solved instances) – be it regarding the number of solved instances or runtime. Only Bitwuzla was able to solve the same number of instances as EP-CBMC.

2. *How do these solvers compare to the back-end solvers used in the commercial tool BTC EP – in particular when considering the default time limit of 60 seconds of BTC EP?*

   EP-CBMC and iSAT3 dominate the four SMT2 solvers if only 60 seconds are available per instance as iSAT3 (8594 solved instances) and EP-CBMC (8269 solved instances) solve within 60 seconds nearly the same number of instances as within 3600 seconds. This is not the case for the SMT2 solvers. It can be observed that the gap to EP-CBMC and iSAT3 increases when decreasing the timeout. The ranking is as follows: Bitwuzla (7748 solved instances), MathSAT5 (6047 solved instances), Z3 (4660 solved instances) and CVC4 (3815 solved instances). It can be expected that these numbers improve when splitting the BMC and k-induction formulas to separate instances for the SMT2 solvers – but even under optimal conditions no SMT2 solver will solve as many instances as EP-CBMC or iSAT3.

To summarize, off-the-shelf SMT solvers are currently unable to outperform the back-end solvers used within BTC EP – in fact it is vice versa, i.e. the two back-end solvers EP-CBMC and iSAT3 outperform Z3, CVC4, MathSAT5 and Bitwuzla on floating-point dominated automotive code.

Among the four SMT2 solvers Bitwuzla clearly outperforms the other three – but is still behind EP-CBMC and iSAT3 when considering practically relevant timeouts. On the other hand, with further optimizations, Bitwuzla might become a promising candidate to supplement our solver portfolio in future versions of BTC EP.

References

[1] L. Westhofen, P. Berger, and J. Katoen, "Benchmarking Software Model Checkers on Automotive Code," in *NFM 2020*, ser. Lecture Notes in Computer Science, R. Lee, S. Jha, and A. Mavridou, Eds., vol. 12229. Springer, pp. 133–150.

[2] ISO, "Road vehicles – Functional safety," 2011.

[3] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A Practical Tutorial on Modified Condition/Decision Coverage," Tech. Rep., 2001.

[4] F. Neubauer, K. Scheibler, B. Becker, A. Mahdi, M. Fränzle, T. Teige, T. Bienmüller, and D. Fehrer, "Accurate Dead Code Detection in Embedded C Code by Arithmetic Constraint Solving," in *SC-square 2016*, ser. CEUR Workshop Proceedings, E. Ábrahám, J. H. Davenport, and P. Fontaine, Eds., vol. 1804. CEUR-WS.org, pp. 32–38.

[5] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS 2004*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, pp. 168–176.

[6] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller, “Incremental Bounded Model Checking for Embedded Software,” *Formal Aspects Comput.*, vol. 29, no. 5, pp. 911–931, 2017.

[7] K. Scheibler, “Applying CDCL to Verification and Test: When Laziness Pays Off,” Ph.D. dissertation, University of Freiburg, Freiburg im Breisgau, Germany, 2017.

[8] T. Bienmüller and T. Teige, “Satisfaction Meets Practice and Confidence,” in *SC-square 2016*, ser. CEUR Workshop Proceedings, E. Ábrahám, J. H. Davenport, and P. Fontaine, Eds., vol. 1804. CEUR-WS.org, pp. 4–7.

[9] D. Beyer, M. Dangl, and P. Wendler, “A unifying view on smt-based software verification,” *J. Autom. Reason.*, vol. 60, no. 3, pp. 299–335, 2018.

[10] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, “Symbolic Model Checking Using SAT Procedures instead of BDDs,” in *DAC 1999*.

[11] N. Eén and N. Sörensson, “Temporal Induction by Incremental SAT Solving,” *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003.

[12] K. L. McMillan, “Interpolation and SAT-Based Model Checking,” in *CAV 2003*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, pp. 1–13.

[13] A. R. Bradley, “SAT-Based Model Checking without Unrolling,” in *VMCAI 2011*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, pp. 70–87.

[14] N. Eén, A. Mishchenko, and R. K. Brayton, “Efficient Implementation of Property Directed Reachability,” in *FMCAD 2011*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., pp. 125–134.

[15] K. Scheibler, F. Winterer, T. Seufert, T. Teige, C. Scholl, and B. Becker, “ICP and IC3,” in *2021 Design, Automation & Test in Europe Conference & Exhibition, DATE 2021*.

[16] F. Winterer, T. Seufert, K. Scheibler, T. Teige, C. Scholl, and B. Becker, “ICP and IC3 with Stronger Generalization,” in *24. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2021*.

[17] Z. Fu and Z. Su, “XSat: A Fast Floating-Point Satisfiability Solver,” in *CAV 2016*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9780. Springer, pp. 187–209.

[18] M. A. Ben Khadra, D. Stoffel, and W. Kunz, “goSAT: Floating-Point Satisfiability as Global Optimization,” in *FMCAD 2017*, D. Stewart and G. Weissenbacher, Eds. IEEE, pp. 11–14.

[19] L. M. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *TACAS 2008*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, pp. 337–340.

[20] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *CAV 2011*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, pp. 171–177.

[21] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, “The MathSAT5 SMT Solver,” in *Proceedings of TACAS*, ser. LNCS, N. Piterman and S. Smolka, Eds., vol. 7795. Springer, 2013.

[22] A. Niemetz and M. Preiner, “Bitwuzla at the SMT-COMP 2020,” *CoRR*, vol. abs/2006.01621, 2020. [Online]. Available: https://arxiv.org/abs/2006.01621

[23] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, “Btor2 , BtorMC and Boolector 3.0,” in *CAV 2018*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, pp. 587–595.

[24] C. Herde, “Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure: Proof Engines for the Analysis of Hybrid Discrete-Continuous Systems,” Ph.D. dissertation, Carl von Ossietzky University of Oldenburg, 2011.

[25] T. Teige, “Stochastic Satisfiability Modulo Theories: A Symbolic Technique for The Analysis of Probabilistic Hybrid Systems,” Ph.D. dissertation, Carl von Ossietzky University of Oldenburg, 2012.

[26] S. Kupferschmid, “Über Craigsche Interpolation und deren Anwendung in der formalen Modellprüfung,” Ph.D. dissertation, University of Freiburg, 2013.

[27] A. Eggers, “Direct Handling of Ordinary Differential Equations in Constraint-Solving-Based Analysis of Hybrid Systems,” Ph.D. dissertation, Carl von Ossietzky University of Oldenburg, 2014.

[28] K. Scheibler, F. Neubauer, A. Mahdi, M. Fränzle, T. Teige, T. Bienmüller, D. Fehrer, and B. Becker, “Accurate ICP-based Floating-Point Reasoning,” in *FMCAD 2016*, R. Piskac and M. Talupur, Eds.

[29] M. Brain, F. Schanda, and Y. Sun, “Building better bit-blasting for floating-point problems,” in *TACAS 2019*, ser. Lecture Notes in Computer Science, T. Vojnar and L. Zhang, Eds., vol. 11427. Springer, pp. 79–98.

[30] C. Barrett, H. Barbosa, M. Brain, A. Irfan, M. Mann, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, A. Wilsond, and Y. Zohar, “CVC4 at the SMT competition 2020.”

[31] A. Biere, “CaDiCaL at the SAT Race 2019,” in *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2019-1. University of Helsinki, pp. 8–9.

[32] M. Brain, V. D’Silva, A. Griggio, L. Haller, and D. Kroening, “Deciding Floating-Point Logic with Abstract Conflict Driven Clause Learning,” *Formal Methods in System Design*, vol. 45, no. 2, pp. 213–245, 2014.