Successful Use of Incremental BMC
in the Automotive Industry

Peter Schrammel[1], Daniel Kroening[1], Martin Brain[1], Ruben Martins[1(✉)],
Tino Teige[2], and Tom Bienmüller[2]

[1] University of Oxford, Oxford, England
[2] BTC Embedded Systems AG, Oldenburg, Germany
ruben.martins@cs.ox.ac.uk, {first.lastname}@cs.ox.ac.uk,
{first.lastname}@btc-es.de

Abstract. Program analysis is on the brink of mainstream usage in embedded systems development. Formal verification of behavioural requirements, finding runtime errors and automated test case generation are some of the most common applications of automated verification tools based on Bounded Model Checking (BMC). Existing industrial tools for embedded software use an off-the-shelf Bounded Model Checker and apply it iteratively to verify the program with an increasing number of unwindings. This approach unnecessarily wastes time repeating work that has already been done and fails to exploit the power of incremental SAT solving. This paper reports on the extension of the software model checker CBMC to support *incremental BMC* and its successful integration with the industrial embedded software verification tool BTC EMBEDDEDTESTER. We present an extensive evaluation over large industrial embedded programs, mainly from automotive industry. We show that incremental BMC cuts runtimes by *one order of magnitude* in comparison to the standard non-incremental approach, enabling the application of formal verification to large and complex embedded software.

1 Introduction

Recent trend estimation [14] in automotive embedded systems revealed ever growing complexity of computer systems, providing increased safety, efficiency and entertainment satisfaction. Hence, automated design tools are vital for managing this complexity and supporting the verification processes in order to satisfy the high safety requirements stipulated by safety standards and regulations. Similar to the developments in hardware verification in the 1990s, verification tools for embedded software are becoming indispensable in industrial practice for hunting runtime bugs, checking functional properties and test suite generation [13]. For example, the automotive safety standard ISO 26262 [22] requires the test suite to satisfy modified condition/decision coverage [18] – a goal that is laborious to achieve without support by a model checker that identifies unreachable test goals and suggests test vectors for difficult-to-reach test goals.

In this paper, we focus on the application of Bounded Model Checking (BMC) to this problem. The technique is highly accurate (no false alarms) and is furthermore able

to generate counterexamples that aid debugging and serve as test vectors. The spiralling power of SAT solvers has made this technique scale to reasonably large programs and has enabled industrial application.

In BMC, the property of interest is checked for traces that execute loops up to a given number of times k. Since the value of k that is required to find a bug is not known a-priori, one has to try increasingly larger values of k until a bug is found. The analysis is aborted when memory and runtime limits are exceeded.[1]

Industrial verification tools based on BMC, such as BTC EMBEDDEDTESTER, use an off-the-shelf Bounded Model Checker and, without additional information about the program to be checked, apply it in an iterative fashion:

```
k=0
while true do
  if BMC(program,k) fails then
    return counterexample
  fi
  k++
od
```

This basic procedure offers scope for improvement. In particular, note that the Bounded Model Checker has to redo the work of generating and solving the SAT formula for time frames 0 to k when called to check time frame $k + 1$. It is desirable to perform the verification *incrementally* for iteration $k + 1$ by building upon the work done for iteration k.

Incremental BMC has been applied successfully to the verification of hardware designs, and has been reported to yield substantial speedups [33,11]. Fortunately, the typical control-loop structure of embedded software resembles the monolithic transition relation of hardware designs, and thus strongly suggests incremental verification of successive loop unwindings. However – to our knowledge – none of the software model checkers for C programs that have competed in the TACAS 2014 Software Verification Competition implement such a technique that ultimately exploits the full power of incremental SAT solving [35,10].

Contributions. The primary contribution of this paper is *experimental*. We quantify the benefit of incremental BMC in the context of the verification of industrial embedded software. To this end,

(1) we survey the requirements for state-of-the-art embedded software verification tools, briefly summarise the underlying theory of the used techniques, and highlight the challenges faced when applying them to industrial code;
(2) we present the first industrial-strength implementation of incremental BMC in a software model checker for ANSI-C programs combining symbolic execution, slicing and incremental SAT solving;
(3) we report on the successful integration of our incremental Bounded Model Checker in the industrial embedded software verification tools BTC EMBEDDEDTESTER

---

[1] One can stop unwinding when the *completeness threshold* [24] of the system is reached, but this threshold is often impractically large.

and EMBEDDEDVALIDATOR where it is used by several hundred industrial users since version 3.4 and 4.3, respectively; and

(4) we give a comprehensive experimental evaluation over a large set of industrial embedded benchmarks, from mainly automotive origin, that quantify the performance gain due to the incremental approach in a BMC-based tool: incremental BMC outperforms the winner of the TACAS 2014 Software Verification Competition [25] by one order of magnitude.

2   Verification of Model-Based Embedded Software

Recent safety standards, e.g. ISO-26262 [22]), cover model-based development and testing techniques for early simulation, testing and verification, and recommend back-to-back testing for showing simulation equivalence between a high-level model and corresponding production code. In the automotive industry, model-based development including automatic code generation is well-established. In particular, SIMULINK for functional modelling and TARGETLINK[2] for automatic code generation from these models are prominent representatives. SIMULINK DESIGNVERIFIER,[3] BTC EMBED-DEDTESTER,[4] REACTIS,[5] and RT-TESTER[6] are examples of tools that complement the software development tool chain for formal verification of safety requirements against design models. These tools are also used for testing, namely, requirement-based and back-to-back testing, including automatic test vector generation for structural coverage criteria.

2.1   Requirements and Challenges

In the above setting, embedded software verification tools have two main applications: (1) proving/disproving safety properties, and (2) covering test goals or proving their unreachability. BMC-based verification engines are a perfect fit for both applications because they can be used to find counterexamples and prove properties by k-induction.

Embedded C code has to meet many conflicting requirements like real-time constraints, low memory footprint and low energy consumption. Code generators offer options to perform certain optimisations towards these goals, often to the detriment of *code size* (and also readability for humans). The observer instrumentation[7] to encode properties and identify the test goals corresponding to code-coverage criteria such as MC/DC [18] produces a non-negligible overhead in the size of the code but introduces little semantic complexity. When using BMC, the size of the SAT formula built from a program further increases whenever internal loops need to be unwound. File sizes of 10 MB and more are common, which poses difficulties to many tools already when

---

[2] http://www.dspace.com/en/pub/home/products/sw/pcgs/targetli.cfm

[3] http://uk.mathworks.com/products/sldesignverifier

[4] http://www.btc-es.de/index.php?lang=2

[5] http://www.reactive-systems.com

[6] https://www.verified.de/products/rt-tester

[7] The observer instrumentation consists of adding a series of flags to the original source code that enables the analysis tool to determine exactly what parts of the code are exercised.

parsing the source code and encoding the program into a SAT formula, mostly due to inefficient data structures. Incremental BMC helps reducing formula sizes and peak memory consumption (see Sec. 4.2) by incremental formula generation and solving.

In practice, many loop unwindings may be needed to detect errors and reach certain tests goals (more than 100 for some of our industrial benchmarks, see Sec. 4.2). *Non-*incremental bounded model checking repeats work such as file parsing, loop unwinding, SAT formula encoding and discards information learnt in the SAT solver every time it is called and so gives away an enormous amount of performance. This effect exacerbates the cost of large unwinding limits that may be needed.

The main challenge addressed by this paper is to exploit all the benefits of incrementality in BMC and to significantly enhance performance of its integration with an industrial-strength embedded verification and test-vector generation tool, namely BTC EMBEDDEDVALIDATOR and EMBEDDEDTESTER. The impact of this successful technology transfer is demonstrated on original industrial embedded software.

2.2   Case Study: Fault-Tolerant Fuel Control System

In this paper, we focus on the verification of C code generated from SIMULINK models. To this end, we illustrate the characteristics of this verification problem with the help of a well-known case study and explain the workflow and principal techniques that a state-of-the-art embedded software verification tool uses.

The Fault-Tolerant Fuel Control System[8] (FUELSYS) for a gasoline engine is representative of a variety of automotive applications as it combines discrete control logic with continuous signal flow and thus establishes a hybrid discrete-continuous system. More precisely, the control logic of FUELSYS is implemented by six automata with two to five states each, while the signal flow is further subdivided into three subsystems with a rich variety of SIMULINK/TARGETLINK blocks involving arithmetic, lookup tables, integrators, filters and interpolation (Fig. 1). The system is designed to keep the air-fuel ratio nearly constant depending on the inputs given by a throttle sensor, a speed sensor, an oxygen sensor (EGO) and a pressure sensor (MAP). Moreover it is tolerant to individual sensor faults and is designed to be highly robust, i.e. after detection of a sensor fault the system is dynamically reconfigured.

Properties of Interest.   The key functional property for FUELSYS is how the air-fuel ratio evolves for each of the four sensor-failure scenarios. Simulation-based approaches show that FUELSYS is indeed fault-tolerant in each case of a single failure: the air-fuel ratio can be regulated after a few seconds to about $80\,\%$ of the target ratio. In addition to *functional* testing of industrial embedded software, safety standards call for *structural* testing of the production code before release deployment.

2.3   Structure of Generated Code

Many modelling languages follow the *synchronous programming paradigm* [17], which is well-suited for modelling time-triggered systems, in which tasks (subsystems of the

---

[8] http://www.mathworks.co.uk/help/simulink/examples/modeling-a-fault-tolerant-fuel-control-system.html
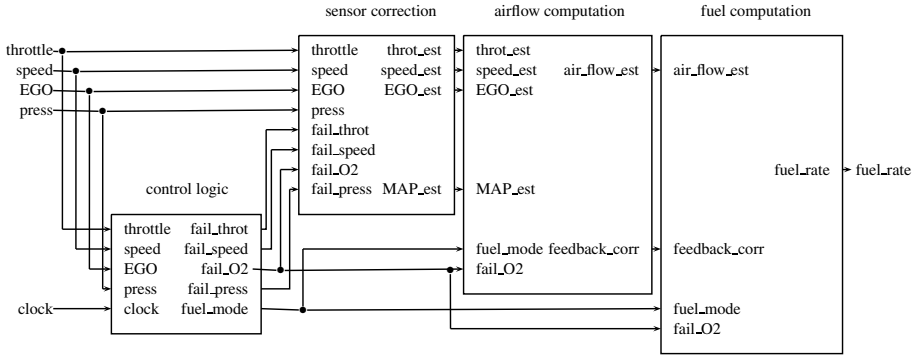
Fig. 1. The SIMULINK Diagram for the Fault-Tolerant Fuel Control System (without the plant model)

model) execute at given rates. Code generation for such languages produces a typical code structure, which corresponds essentially to a non-preemptive operating system task scheduler. Most code generators provide the scheduler for time-triggered execution or code to interface with popular real-time operating systems. In either case, the functionality corresponds to the following pseudo code:

```
1   void main() {
2       state s; inputs i; outputs o;
3       initialize(s);
4       while(true) { //main loop
5           i = read_inputs();
6           (o,s) = compute_step(i,s);
7           write_outputs(o);
8           wait(); //wait for timer interrupt
9       }
10  }
```

The distinguishing characteristic of such a reactive program is its unbounded main loop, which we will analyse incrementally. All other loops contained within that loop, e.g. to iterate over arrays or interpolate values using look-up tables, have a statically bounded number of iterations and can be fully unwound.

2.4   Analysis with BMC and k-Induction

Property Instrumentation. Formal verification requires formalisations of high-level requirements, often using observer Büchi automata with a dedicated 'error state' generated from temporal logic descriptions. Test vector generation is done for code-coverage criteria such as branches, statements, conditions and MC/DC of the production C code. For FUELSYS, for example, MC/DC instrumentation yields 251 test goals. The properties to be verified or tested have in common that they can be reduced to a reachability problem. In formal verification of safety properties, we prove that the error state is unreachable, whereas the aim of test vector generation is to obtain a trace that demonstrates reachability of the goal state.

To validate whether the air-fuel ratio in the FUELSYS controller is regulated after a few seconds to be within some margin of the target ratio, one has to instrument the

reactive program, as sketched above, with an observer implementing the asserted property. For instance, consider the requirement "If some sensor fails for the first time then within 10 seconds the air-fuel ratio will keep in between the range of $80\,\%$ to $120\,\%$ of the target ratio forever." The code fragment for an observer for this requirement may look as follows:

```
1   // detection of first sensor failure
2   if (sensor_fail == 1 && observe_ratio == 0) {
3       // initialize observer variables
4       observe_ratio = 1;
5       counter = 0;
6       violated = 0;
7   }

8   if (observe_ratio == 1) { // observation mode
9       if (counter >= 10 &&
10          (air_fuel_ratio < 0.8*target_ratio ||
11           air_fuel_ratio > 1.2*target_ratio))
12              violated = 1;
13      counter++;
14  }

15  assert(violated == 0); // safety property
```

In order to verify that the above property actually holds, one has to show that the assertion in the observer code is always satisfied. We use BMC for refutation of the assertion, and k-induction for proving it.

Bounded Model Checking. BMC [2] can be used to check the existence of a path $\pi = \langle s_0, s_1, \ldots, s_k \rangle$ of length k between two states s_0 and s_k belonging to sets respectively described by ϕ and ψ. This check is performed by deciding satisfiability of the following formula using a SAT or SMT solver:

$$\phi(s_0) \wedge \bigwedge_{0 \leq j < k} T(s_j, i_j, s_{j+1}) \wedge \psi(s_k) \tag{1}$$

If the solver returns the answer "satisfiable", it also provides a satisfying assignment to the variables $(s_0, i_0, s_1, i_1, \ldots, s_{k-1}, i_{k-1}, s_k)$. The satisfying assignment represents one possible path $\pi = \langle s_0, s_1, \ldots, s_k \rangle$ from ϕ to ψ and identifies the corresponding input sequence $\langle i_0, \ldots, i_{k-1} \rangle$. Hence, BMC is useful for refuting safety properties (where ϕ gives the set of initial states and ψ defines the error states) and generating test vectors (where ψ defines the test goal to be covered).

Unbounded Model Checking by k-Induction. BMC can prove reachability, whereas unreachability can be shown using $k-$induction [31,11,16,7]. The predicate $\neg\psi$ is an (inductive) invariant, i.e., it holds in all reachable states, if each of the following two formulae, base case (BC) and induction step (SC), are unsatisfiable for a given k (assuming that we have already checked for up to $k - 1$):

$$\begin{aligned}
&\text{(BC)} \quad \phi(s_0) \wedge \bigwedge_{0 \leq j < k} \neg\psi(s_j) \wedge T(s_j, i_j, s_{j+1}) \wedge \psi(s_k) \\
&\text{(SC)} \quad \bigwedge_{0 \leq j \leq k} \neg\psi(s_j) \wedge T(s_j, i_j, s_{j+1}) \wedge \psi(s_{k+1})
\end{aligned} \tag{2}$$

The base case checks if the formula is unsatisfiable, when this occurs we say that $\neg\psi$ holds in the first k steps. The induction step checks if we can conclude from the invariant holding over any k consecutive steps that it holds for the $(k + 1)^{st}$ step. If the base step

fails, i.e. above formula is satisfiable and a counterexample is given, we have refuted the property. If it holds and the induction step fails, we do not know whether $\neg\psi$ is invariant. Only if both formulae hold we have proved that $\neg\psi$ is invariant.

Both base step and induction step are essentially instances of BMC: starting from the initial state ϕ for the base case, and starting from *any* state for the induction step. Thus, similar to BMC, k-induction can be applied by using a sequence of increasing values for k.

3   Incremental BMC

In this section, we explain the technical background of incremental SAT solving and how it is employed in our implementation of incremental BMC.

3.1   Incremental SAT Solving

The first ideas for incremental SAT solving date back to the 1990s [21,32]. The question is how to solve a sequence of similar SAT problems while reusing effort spent on solving previous instances, i.e. reusing the internal state and learnt information of the solver. Incremental SAT solving is easy as long as formulas are *growing monotonically*, i.e. clauses are added to the formula. Removing clauses is trickier and requires additional solver features like solving *under assumptions* [11], which is the most popular approach to incremental SAT solving: assumptions are temporary assignments to variables that hold solely for one specific invocation of the SAT solver. In Sec. 3.2, we will explain how SAT solving under assumptions allows us to emulate the removal of clauses.

An alternative approach is to use SMT solvers. SMT solvers offer an interface for pushing and popping clauses in a stack-like manner. Pushing adds clauses, popping removes them from the formula. This makes the modification of the formula intuitive to the user, but the efficiency depends on the underlying implementation of the push and pop operations. For example, in [15] it was observed that some SMT solvers (like Z3) are not optimised for incremental usage and hence perform worse incrementally than non-incrementally.

Since CBMC itself implements powerful bitvector decision procedures, we use the SAT solver MINISAT2 [10] as a backend solver, and focus on solving under assumptions in the sequel.

3.2   Incremental BMC

We will now discuss which aspects have to be taken into account when implementing an incremental approach in a software Bounded Model Checker. We will show that symbolic execution and slicing can be performed without interfering with the requirement of monotonic formula construction for incremental SAT solving, whereas incremental unwinding and transition function refinements require solving under assumptions.

Following the construction in [11] for finite state machines, incremental BMC can be formulated as a sequence of SAT problems $\Phi(k)$ that we need to solve:

$$\begin{aligned} \Phi(0) \quad &:= \phi(s_0) \wedge (\Psi(0) \vee \alpha_0) \\ \Phi(k+1) &:= \Phi(k) \wedge T(s_k, i_k, s_{k+1}) \wedge \alpha_k \wedge (\Psi(k+1) \vee \alpha_{k+1}) \end{aligned} \tag{3}$$

where $\Psi(k)$ is the disjunction $\bigvee_{0 \le j \le k} \psi(s_j)$ of error states ψ to be proved unreachable up to iteration k. This means that the verification fails if *at least one* of the error states is reachable. Since the set of ψ_js grows in each iteration, our problem is not monotonic: one has to *remove $\Psi(k)$* when adding $\Psi(k+1)$ because $\Psi(k)$ subsumes $\Psi(k+1)$.

Here, solving under assumptions comes to rescue. In iteration k, the α_k is assumed to be false, whereas it is assumed true for iterations $k' > k$. This has the effect that in iteration k' the formula $(\Psi(k) \vee \alpha_k)$ becomes trivially satisfied. Hence, it does not contribute to the (un)satisfiability of $\Phi(k')$, which emulates its deletion.[9]

Symbolic Execution. For software (3) results in large formulae and would be highly inefficient for the purpose of BMC. In practice, software model checkers use *symbolic execution* in order to exploit, for example, constant propagation and pruning branches when conditionals are infeasible, while generating the SAT formula and thus reducing its size. This means that the formula describing T is the result of symbolic execution, and that formulae T and Ψ are actually dependent on k. Fortunately, this does not affect the correctness of above formula construction and we can replace T by T_k in (3) and ψ by ψ_k in the definition of $\Psi(k)$. T_k denotes the transition formula obtained by symbolic execution of the kth time frame (i.e. unwinding), and ψ_k the assertions collected for this time frame.

Slicing. Another feature used by state-of-the-art software model checkers is slicing: The purpose of slicing is, again, reducing the size of the SAT formula by removing (or better: not generating) those parts of the formula that have no influence on its satisfiability. There are many techniques how to implement slicing with the desired trade-off between runtime efficiency and its formula pruning effectiveness [34].

Slicing is performed relative to $\Psi(k)$. We said that the number of disjuncts ψ_j in Ψ is growing monotonically with k. Hence, we will show that, assuming that our slicing operator is monotonic, we obtain a monotonic formula construction:

The transition formula for each time frame T_k obtained by symbolic execution is a conjunction $\bigwedge_{\tau \in M} \tau$ of subrelations τ (e.g., formulae corresponding to program instructions). The slicing operator *slice* selects a subset of M. The operator *slice* is monotonic iff $M \subseteq M' \implies slice(M) \subseteq slice(M')$.

We can then view the conjunction of transition relations for k time frames $\widehat{T}(k) = \bigwedge_{0 \le j \le k} T_j$ as $\bigwedge_{\tau \in M_k} \tau$. A slice $\widehat{T}^{sliced}(k)$ of $\widehat{T}(k)$ is $\bigwedge_{\tau \in M'_k} \tau$ where $M'_k \subseteq M_k$. An incremental slice is then defined as the difference between $\widehat{T}^{sliced}(k+1)$ and $\widehat{T}^{sliced}(k)$: $T^{sliced}_{k+1} = \bigwedge_{\tau \in M'_{k+1} \setminus M'_k} \tau$.

Monotonicity of formula construction follows from $M'_{k+1} \subseteq M_{k+1}$ and the assumed monotonicity $M'_k \subseteq M'_{k+1}$ of the slicing operator. We can thus replace T by T^{sliced}_k in (3). Mind that T^{sliced}_k contains also subrelations τ for time steps $k' < k$.

Our slicing operator computes the (syntactic) variable dependency graph for $\widehat{T}(k+1)$ and obtains M'_{k+1} as the set of all τ which $\Psi(k+1)$ depends on. Then only those τ

---

[9] For a large number of iterations k, such trivially satisfied subformulas might accumulate as "garbage" in the formula and slow down its resolution. Restarting the solver at appropriate moments is the common solution to this issue.

in M'_{k+1} are added to the formula that have not been in the slice for the previous time frame, resulting in T^{sliced}_{k+1}.

Refinements. Incremental SAT solving is also used for incremental refinements of the transition relation T for bitvectors [4,8] and arrays [28], for example. Applying bitvectors and arrays refinements inside an incremental software Bounded Model Checker requires using several incremental formula encodings for (in general, non-monotonic) refinements. These refinements are global over all unwindings, so that in iteration k we have to further refine transition relations $T_{k'}$ from earlier iterations $k' < k$. For details on the formula construction for refinements inside an incremental Bounded Model Checker we refer to the extended version of the paper [30].

4   Experimental Evaluation

We present the results of our experimental evaluation of incremental BMC and incremental k-induction on industrial programs from mainly automotive origin. The goal of this evaluation is to quantify the benefit from an incremental approach in a BMC-based tool infrastructure.[10] The experiments for this study were performed on a 3.5 GHz Intel Xeon machine with 32 GB of physical memory running Windows 7 with a time limit of 3,600 seconds.

4.1   Implementation

We have implemented our extension[11] for incremental BMC in the Bounded Model Checker for ANSI-C programs CBMC [6] using the SAT solver MINISAT2 [10]. Incremental CBMC can be used with specific options that enables extra features, namely: (i) slicing, (ii) preprocessing, and (iii) formula-level refinements. The goal of these techniques is to reduce the size of the SAT formula that is being generated. Slicing reduces the size of the SAT formula by eliminating irrelevant paths of the program. Preprocessing through the MINISAT2 simplifier reduces the size of the SAT formula after it has been generated, and formula-level refinements performs an incremental build of the SAT formula. For information regarding the command line options of incremental CBMC we refer to the CPROVER wiki page.[12]

In the integration of CBMC with BTC EMBEDDEDTESTER and EMBEDDEDVALIDATOR, a master routine selects the next verification/test goal to be analysed starting from instrumented C code. After some preprocessing like source-level slicing and internal-loop unwinding the resulting reachability task is given to CBMC. If CBMC is able to solve the problem within the user-defined time limit, the result, i.e. bounded or unbounded unreachability, or a counterexample in case of reachability, is reported back

---

[10] For a comparison with alternative verification approaches, we kindly refer to the results of the Software Verification Competition (http://sv-comp.sosy-lab.org), where BMC-based tools rank in the top 3 every year.

[11] Source code available from http://www.cprover.org/svn/cbmc/branches/peter-incremental-unwinding

[12] http://www.cprover.org/wiki/doku.php?id=how_to_use_incremental_unwinding

Table 1. Benchmark characteristics from industrial programs

		LOC	operators			input variables			state variables			observer	
			cond	mul	div/rem	bool	int	float	bool	int	float	bool	unwindings
SAT	max	31222	17103	669	75	688	477	189	3876	750	107	22	106
	average	7572	4306	188	9	103	79	19	583	136	15	9	22
UNSAT	max	23014	49530	567	37467	212	282	188	708	663	32	22	10
	average	4854	6014	160	1257	30	51	9	163	73	3	7	10

to the master process. Otherwise, i.e. in case of a timeout, CBMC is killed but information about the solved unwindings of the reactive main loop is given back, which frequently is a useful result for the user since it may indicate the absence of shallow bugs.

To prove unreachability of verification/test goals (properties), k-induction is performed (see Sec. 2.4). For this purpose BTC EMBEDDEDTESTER generates two source files, one containing the base case, which is a normal BMC problem with the property given as assertion (cf. Equ. (2) (BC)); the file for the step case havocs variables modified in the loop and the invariant property is assumed at the beginning of the loop and asserted at the end of the loop (cf. Equ. (2) (SC)). To check the step case, we require a reversed termination behaviour of CBMC, i.e. it continues unwinding as long as the problem is SAT and stops as soon as it is UNSAT.

4.2 Incremental BMC for Embedded Software

We report results on industrial programs for the integration of CBMC with BTC EMBEDDEDTESTER and EMBEDDEDVALIDATOR. For these experiments, we used 60 industrial benchmarks, which are original, unmodified code from BTC customers, mainly from automotive applications. Unfortunately, software in the automotive domain is closed source, and hence, being subject to NDAs, these benchmarks cannot be made public.[13] These benchmarks have the property of having only one unbounded loop. Half of the benchmarks are bug-free (UNSAT instances), half contain a bug (SAT instances). This benchmark suite is an indicator for performance of model checking tools in an industrial setting as it covers a representative spectrum of embedded software.

A summary of the benchmark characteristics is listed in Table 1. Besides the number of lines of code, we give the number of conditional operators, multiplications and divisions or remainder operations, which are a good indicator for the difficulty of the benchmark, because they generate large formulae — recall that for each "/" occurring in the program, CBMC has to generate a divider circuit. The surprisingly high number of conditional operators in most of the benchmarks is due to the preprocessing of conditional assignments by BTC EMBEDDEDTESTER and hints at the amount of branching in these benchmarks. Moreover, we list the number of input and state variables, and the variables introduced by the observer instrumentation.

Runtimes. We compared the incremental (i) with the non-incremental (ni) approach and evaluated the impact of slicing (s), SAT preprocessing (p) and bitvector refinement (r).[14] The incremental and non-incremental approaches were compared by activating

---

[13] To mitigate this problem, we present a detailed summary of the benchmark characteristics in the extended version of the paper [30].

[14] Array refinement is not used because the benchmarks do not contain arrays.
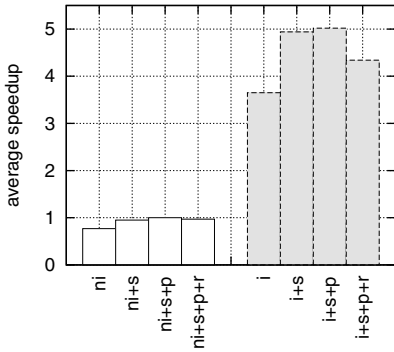
(a) Effect of slicing, SAT formula preprocessing and bitvector refinement
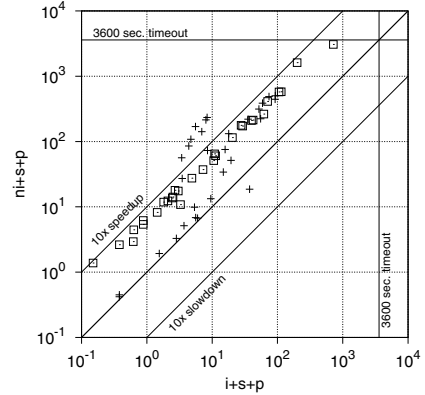
(b) Comparison between ni+s+p and i+s+p
(+ SAT instances; □ UNSAT instances)

Fig. 2. Incremental vs. non-incremental BMC

none of the three techniques, with slicing only (+s), with slicing and preprocessing (+s+p), and with all three options activated (+s+p+r). The maximum number of loop unwindings was fixed to 10 for the UNSAT instances in order to balance a significant exploration depth with reasonable analysis runtimes. For SAT instances, a maximum number of loop unwindings was not fixed since the incremental and non-incremental approaches are bound to terminate when the unwinding depth reaches the depth of the bug. The number of unwindings are listed in the last column in Table 1.

Fig. 2 shows the comparison between the incremental and non-incremental approaches and the impact of each tool option on their performance. Fig. 2a shows the average geometric mean [12] speedup of instances that were solved by all approaches. We consider as baseline the (ni+s+p) approach since it was the best non-incremental approach. Each bar shows the average geometric mean speedup of each approach when compared to (ni+s+p). For example, (ni) has a speedup of 0.77, i.e. (ni) is on average $0.77\times$ slower than (ni+s+p). On the other hand, all incremental versions are much faster than the non-incremental versions. For example, (i) is on average over $3.5\times$ faster than (ni+s+p) and (i+s+p) is on average over $5\times$ faster than (ni+s+p). We observe the following effects of the tool options: (i) slicing shows significant benefits overall (also on peak memory consumption); (ii) not using formula preprocessing is a bad idea in general; and (iii) bitvector refinement shows benefits for UNSAT instances, but produces overhead for SAT instances which deteriorates the overall performance of the tool (see the extended version of the paper [30] for more details). Even though the tool options have some positive effects, they are rather minor in comparison to the performance gains from using an incremental approach.

Since the best incremental and non-incremental approaches were obtained with the configuration (+s+p), we will use this configuration for both approaches on the results described in the remainder of the paper.

Fig. 2b shows a scatter plot with runtimes of the best non-incremental (ni+s+p) and incremental (i+s+p) approaches. Each point in the plot corresponds to an instance, where the x-axis corresponds to the runtime required by the incremental approach and the y-axis corresponds to the runtime required by the non-incremental approach. If an instance is above the diagonal, then it means that the incremental approach is faster than the non-incremental approach, otherwise it means that the non-incremental approach is faster. SAT instances are plotted as crosses, whereas UNSAT instances are plotted as squares. Incremental BMC significantly outperforms non-incremental BMC. For SAT instances, the advantage of incremental BMC is negligible for the easy instances, whereas speedups are around a factor of 10 for the medium and hard instances. For UNSAT instances, speedups are also significant and most instances have a speedup of more than a factor of 5.

Solving vs. Overall Runtime. Since CBMC is used as a black-box with BTC EM-BEDDEDTESTER and EMBEDDEDVALIDATOR, the non-incremental approach has to re-parse files in each iteration. One might argue that removing this overhead is the main reason for the speedup observed. However, the overhead for parsing files, symbolic execution and slicing when compared to generating and solving SAT formula is similar for the incremental and non-incremental approach. The incremental approach spends 27% of its time solving the SAT formula (582 out of 2,151 seconds), whereas the non-incremental approach spends 28% of its time (3,317 out of 11,811 seconds). Unsurprisingly, solving the instance for the largest k in the non-incremental approach takes a considerable amount of time (around 24%), when compared to the total time for solving the SAT formulae for iterations 1 to k (784 out of 3,317 seconds).

An explanation for these speedups might be the size of the queries issued in both approaches. The average number of clauses per solver call is halved from 1,367k clauses for the non-incremental approach to 709k clauses for the incremental approach. Similarly, the average number of variables is less than a third in the incremental approach when compared to the non-incremental approach, being 217k and 746k respectively.

Smaller query sizes also have an effect on peak memory consumption which is reduced by 30% for UNSAT benchmarks; for SAT benchmarks, however, we observed a 10% increase.

4.3   Code Coverage on FUELSYS Using BTC EMBEDDEDTESTER

As reported in the previous section, enabling CBMC to work incrementally led to tremendous performance gains. In order to assess whether these improvements have practical impact in the *integration* of CBMC with an industrial-strength test-vector generation tool, we compared the performance of BTC EMBEDDEDTESTER with the incremental feature of CBMC being disabled and enabled. The time limit per subtask was 10 minutes and the unwinding depth for all internal loops was 50. For unwinding depth 10 of the main loop, the incremental feature improves the overall runtime from 152.3 to 70.4 minutes, i.e. more than $2\times$ faster, and for unwinding depth 50 from 377.4 to 108.5 minutes, i.e. more than $3\times$ faster.

4.4   Incremental k-Induction for Embedded Software

To compare the performance of incremental and non-incremental approaches for k-induction, we considered the subset of UNSAT benchmarks for which k-induction required more than 1 iteration (see the extended version of the paper [30] for more details). Note that when k-induction requires only 1 iteration, the performance of both approaches is similar.

Fig. 3 shows a scatter plot with the runtimes of incremental and non-incremental k-induction using the tool options (+s+p). Instances that correspond to the base case are plotted as crosses, whereas instances that correspond to the step case are plotted as squares. The runtimes for

Fig. 3. Incremental k-induction
(+ BC instances; □ SC instances)

both incremental and non-incremental checking are relatively small. These are due to the small number of iterations required by k-induction to prove the unreachability of the properties present on these benchmarks (between 2 and 4 iterations with an average of 2.4 iterations per instance). Incremental checking is on average $2\times$ faster than non-incremental checking, on both base and step cases.
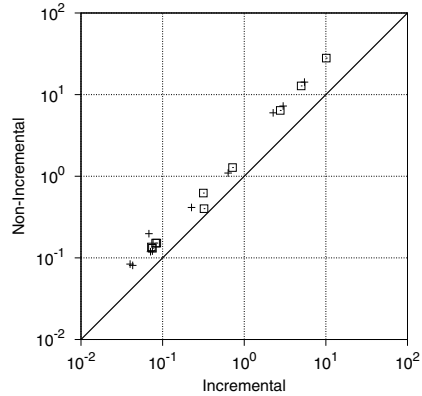
5   Related Work

Most related is recent work on a prototype tool NBIS [15] implementing incremental BMC using SMT solvers. They show the advantages of incremental software BMC. However, they do not consider industrial embedded software and have evaluated their tool only on small benchmarks that are very easy for both, incremental and non-incremental, approaches (runtimes <1s).[15]

Bit-precise formal verification techniques are indispensable for embedded system models and implementations, that have low-level, i.e. C language, semantics like discrete-time SIMULINK models. The importance of this topic has recently attracted attention as shown by publications on verification using SMT Solving [19,26], test case generation [27], symbolic analysis for improving simulation coverage [1], and directed random testing [29]. Yet, all these works have not exploited incremental BMC.

The test vector generation tool FSHELL [20] uses incremental SAT solving to check the reachability of a set of test goals. However, it assumes a fixed unwinding of the loops. There is no reason why incremental BMC should not boost its performance when increasing loop unwindings need to be considered. Test vector generation tools like KLEE [5] use incremental SAT solving to extend the paths to be explored. However, they consider only single paths at a time, whereas BMC explores all paths simultaneously.

---

[15] Unfortunately, a working version of the tool was not available at time of submission.

Incremental SAT solving has important applications in other verification techniques like the IC3 algorithm [3,9] and incremental BMC is standard for hardware verification [23,36]. We show that the speedups of incremental SAT solving reported in [11] regarding k-induction on small HW circuits carry over to industrial embedded software.

6 Conclusions

We claim that incremental BMC is an indispensable technique for industrial embedded software verification based on BMC. To underpin this claim, we report on the successful integration of our incremental extension of CBMC into an industrial embedded software verification tool. Our experiments demonstrate one-order-of-magnitude speedups from incremental approaches on industrial embedded software benchmarks for BMC and k-induction. These performance gains result in faster property verification and higher test coverage, and thus, a productivity increase in embedded software verification.

Incremental BMC is effective on embedded software because of its specific properties (one big unbounded loop, whereas other loops are bounded). Nonetheless, we can also expect benefits for general software where loops and control structures are more irregular. A preliminary report on incremental BMC for programs with *multiple loops* is presented in the extended version of the paper [30]. Even though the current approach for multiple loops can still be improved, we already observe significant speedups that show the applicability of incremental BMC beyond embedded software.

References

1. Alur, R., Kanade, A., Ramesh, S., Shashidhar, K.C.: Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In: EMSOFT, pp. 89–98 (2008)
2. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
3. Bradley, A.R.: IC3 and beyond: Incremental, Inductive Verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, p. 4. Springer, Heidelberg (2012)
4. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: Deciding Bit-Vector Arithmetic with Abstraction. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007)
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: OSDI, pp. 209–224 (2008)
6. Clarke, E., Kroning, 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)
7. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software Verification Using k-Induction. In: Yahav, E. (ed.) Static Analysis. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011)
8. Eén, N., Mishchenko, A., Amla, N.: A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In: FMCAD, pp. 181–188 (2010)
9. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD, pp. 125–134 (2011)

10. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)

11. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. ENTCS 89(4), 543–560 (2003)

12. Fleming, P., Wallace, J.: How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. CACM 29(3), 218–221 (1986)

13. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. STVR 19(3), 215–261 (2009)

14. Gunnarsson, D., Kuntz, S., Farrall, G., Iwai, A., Ernst, R.: Trends in automotive embedded systems. In: CODES+ISSS, pp. 9–10 (2012)

15. Günther, H., Weissenbacher, G.: Incremental bounded software model checking. In: SPIN, pp. 40–47 (2014)

16. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: FMCAD, pp. 1–9 (2008)

17. Halbwachs, N.: Synchronous programming of reactive systems. Kluwer (1993)

18. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A practical tutorial on modified condition/decision coverage. Tech. rep., NASA (May 2001)

19. Herber, P., Reicherdt, R., Bittner, P.: Bit-precise formal verification of discrete-time MATLAB/Simulink models using SMT solving. In: EMSOFT, pp. 1–10 (2013)

20. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 151–166. Springer, Heidelberg (2009)

21. Hooker, J.N.: Solving the incremental satisfiability problem. JLP 15(1&2), 177–186 (1993)

22. ISO 26262: Road vehicles – Functional safety (2011)

23. Jin, H., Somenzi, F.: An incremental algorithm to check satisfiability for bounded model checking. ENTCS 119(2), 51–65 (2005)

24. Kroning, D., Strichman, O.: Efficient computation of recurrence diameters. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 298–309. Springer, Heidelberg (2002)

25. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker – (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014)

26. Manamcheri, K., Mitra, S., Bak, S., Caccamo, M.: A step towards verification and synthesis from Simulink/Stateflow models. In: HSCC, pp. 317–318 (2011)

27. Peranandam, P., Raviram, S., Satpathy, M., Yeolekar, A., Gadkari, A.A., Ramesh, S.: An integrated test generation tool for enhanced coverage of Simulink/Stateflow models. In: DATE, pp. 308–311 (2012)

28. Pnueli, A., Strichman, O.: Reduced functional consistency of uninterpreted functions. ENTCS 144(2), 53–65 (2006)

29. Satpathy, M., Yeolekar, A., Ramesh, S.: Randomized directed testing (REDIRECT) for Simulink/Stateflow models. In: EMSOFT, pp. 217–226 (2008)

30. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Incremental bounded model checking for embedded software (extended version). CoRR abs/1409.5872 (2014), http://arxiv.org/abs/1409.5872

31. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)

32. Silva, J.M., Sakallah, K.A.: Robust search algorithms for test pattern generation. In: FTCS, pp. 152–161 (1997)
33. Shtrichman, O.: Pruning techniques for the SAT-based bounded model checking problem. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 58–70. Springer, Heidelberg (2001)
34. Tip, F.: A survey of program slicing techniques. Tech. rep., CWI-Amsterdam (1994)
35. Whittemore, J., Kim, J., Sakallah, K.A.: SATIRE: A new incremental satisfiability engine. In: DAC, pp. 542–545 (2001)
36. Wieringa, S.: On incremental satisfiability and bounded model checking. In: Design & Impl. of Formal Tools & Sys., pp. 46–54 (2011)