DOI 10.1007/s00165-017-0419-1

Formal Aspects of Computing (2017) 29: 911–931

Formal Aspects of Computing

Incremental bounded model checking for embedded software

Peter Schrammel[1,2], Daniel Kroening[2], Martin Brain[2], Ruben Martins[2,3], Tino Teige[4] and Tom Bienmüller[4]

[1] School of Engineering and Informatics, University of Sussex, Brighton, BN1 9RH, UK
[2] Department of Computer Science, University of Oxford, Oxford, UK
[3] Department of Computer Science, University of Texas at Austin, Austin, USA
[4] BTC Embedded Systems AG, Oldenburg, Germany

Abstract. Program analysis is on the brink of mainstream usage in embedded systems development. Formal verification of behavioural requirements, finding runtime errors and 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 article 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 the 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. We furthermore report promising results on analysing programs with arbitrary loop structure using incremental BMC, demonstrating its applicability and potential to verify general software beyond the embedded domain.

Keywords: Embedded systems, Bounded model checking, Incremental SAT solving, k-induction

1. Introduction

Recent trend estimation [GKF+12] in automotive embedded systems indicates 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 [FWA09]. For example, the automotive safety standard ISO 26262 [ISO11] requires the test suite to satisfy modified condition/decision coverage [HVCR01] – 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.

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under Grant Agreement Number 295311 "VeTeSS" and ERC Project 280053 "CPROVER".
Correspondence and offprint requests to: P. Schrammel, e-mail: p.schrammel@sussex.ac.uk

In this article, 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 increasing 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 [Str01, ES03b]. 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 recent Software Verification Competitions implement such technique that ultimately exploits the full power of incremental SAT solving [WKS01, ES03a].

Contributions. The primary contribution of this article is mainly *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 and EMBEDDEDVALIDATOR where it is used by several hundred industrial users since version 3.4 and 4.3, respectively;
4. we give a comprehensive experimental evaluation over a large set of industrial embedded benchmarks, mainly from the automotive industry, 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 [KT14] by one order of magnitude;
5. we formulate the encoding of the incremental BMC problem as a system of recurrence equations, and extend it to include incremental formula refinements; and
6. in order to demonstrate the potential of incremental BMC for general, non-embedded programs, we implement two loop unwinding strategies for handling programs with multiple loops incrementally and compare their performance on benchmarks from the Software Verification Competition.

This article is an extended version of the paper [SKB+15] and extends it with contributions (5) and (6).

[1] One can stop unwinding when the *completeness threshold* [KS03, KOS+11] of the system is reached, but this threshold is often impractically large.

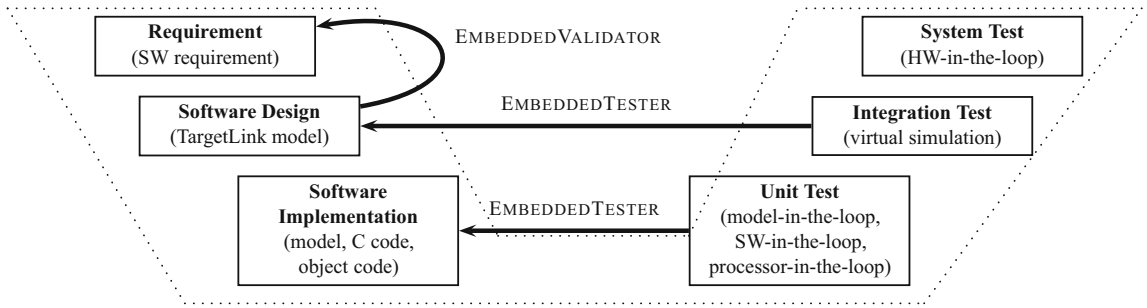


Fig. 1. Tool chain for embedded software development in the V model

2. Verification of model-based embedded software

Recent safety standards, e.g. ISO-26262 [ISO11], 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[2] for functional modelling and TARGETLINK[3] for automatic code generation from these models are prominent representatives. SIMULINK DESIGNVERIFIER,[4] BTC EMBEDDEDTESTER,[5] REACTIS,[6] and RT-TESTER[7] are exemplars 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.[8]

An example of an embedding of this tool chain into the V model, the software development model suggested by ISO-26262, is illustrated in Fig. 1. Tools such as BTC EMBEDDEDVALIDATOR support the automation of formally verifying the requirements against the design model. On lower levels, automated test generation tools such as BTC EMBEDDEDTESTER help validate the implementation in the unit and integration test phases.

In this article, we focus on the verification of C code generated from these models. To this end, we illustrate the characteristics of this verification problem with the help of a well-known case study (Sect. 2.2) and explain the workflow and principal techniques that a state-of-the-art verification tool for embedded software uses.

2.1. Requirements and challenges

In the setting above, 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. Fig. 2 illustrates the schematic architecture of such tools. They consist of a frontend that interacts with the user and a verification backend that performs the actual analysis. To achieve good usability of such a tool, it is important to hide the underlying technical details of the verification backend from the user.

Verification tools, such as BTC EMBEDDEDVALIDATOR, target application (1). They take as inputs the source code (or a design model) and a specification, typically a set of predefined properties, e.g. to check for common runtime errors such as overflows or division-by-zero, or user-defined properties that formalise functional requirements. The properties are then instrumented into the source code (or design model), typically on the level of an intermediate representation. We will give an illustrative example for such an instrumentation in Sect. 2.4. The instrumented code is then checked by a model checker. The frontend reports to the user whether properties have been proved or disproved. In the latter case, the model checker provides a counterexample that is reported to the user for debugging.

[2] http://www.mathworks.co.uk/products/simulink/.
[3] http://www.dspace.com/en/pub/home/products/sw/pcgs/targetli.cfm.
[4] http://uk.mathworks.com/products/sldesignverifier.
[5] http://www.btc-es.de/index.php?lang=2.
[6] http://www.reactive-systems.com.
[7] https://www.verified.de/products/rt-tester.
[8] The topic of model-based testing methods is discussed in detail in a range of surveys [CRT10, NT10, PdSSM12].

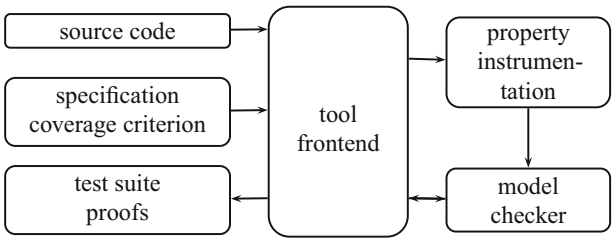


Fig. 2. Typical architecture of a model checker for embedded software

Application (2) is addressed by test generation tools, e.g. BTC EMBEDDEDTESTER. Their input is the source code and a coverage criterion, e.g. MC/DC (Modified Condition/Decision Coverage) [HVCR01]. MC/DC requires that test executions reach not only each function entry and function exit and both outcomes of a decision (both branches of if-then-else), but they also have to show that each basic Boolean condition (that is part of a more complex Boolean decision) independently affects the outcome of the decision. Similar to properties, these coverage criteria are instrumented into the code as test goals whose reachability is to be proven by the model checker. The counterexamples provided by the model checker are then transformed into a test suite and presented to the user.

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[9] to encode properties and identify the test goals corresponding to code-coverage criteria such as MC/DC 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 parsing the source code and encoding the program into a SAT formula, mostly due to inefficient data structures. Incremental BMC helps reduce formula sizes and peak memory consumption (see Sect. 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 Sect. 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 article 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

The Fault-Tolerant Fuel Control System[10] (FUELSYS) for a gasoline engine, originally introduced as a demonstration example for MATLAB SIMULINK/STATEFLOW and then adapted for dSPACE TARGETLINK, is representative of a variety of automotive applications as it combines discrete control logic via STATEFLOW with continuous signal flow expressed by SIMULINK or TARGETLINK 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. 3).

[9] 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.

[10] http://www.mathworks.co.uk/help/simulink/examples/modeling-a-fault-tolerant-fuel-control-system.html.

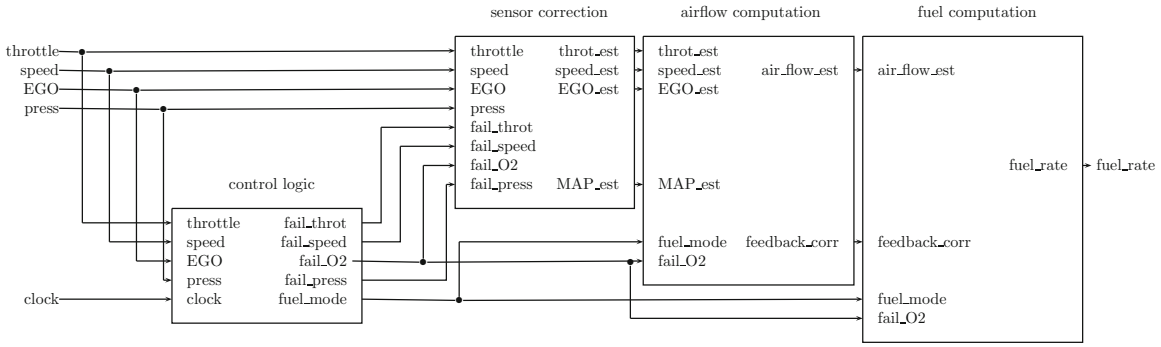


Fig. 3. The SIMULINK diagram for the Fault-Tolerant Fuel Control System (without the plant model)

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 that for each of the four sensor-failure scenarios the air-fuel ratio reaches a given range around a set target ratio within a given bounded time span. 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. In Sect. 2.4, we give a brief overview about such standards and the state of practice of their implementation in industry.

2.3. Structure of generated code

Many modelling languages follow the *synchronous programming paradigm* [Hal93], which is well-suited for modelling time-triggered systems, in which tasks (subsystems of the 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 [Bue62] 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 s the air-fuel ratio will always stay between the range of 80–120% of the target ratio." 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. We model a reactive program, as given in Sect. 2.3, as a transition system with initial states ϕ (function initialize) and a deterministic transition function $T : (s, i) \mapsto s'$ that maps a state s and an input i to a resulting state s' (function compute_step; w.l.o.g. we assume that the outputs are part of the state in order to simplify the notation). BMC [BCCZ99, CBRZ01] 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:

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

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). In the latter case, the initial state s_0 together with the input sequence $\langle i_0, \ldots, i_{k-1} \rangle$ is a test vector.

Unbounded Model Checking by k-Induction. BMC can prove reachability, whereas unreachability can be shown using induction. Let us first define the notion of an invariant. 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 valid.

$$\begin{array}{ll} \text{(BC)} & \forall\, s : \phi(s) \Longrightarrow \neg\psi(s) \\ \text{(SC)} & \forall\, s, i, s' : \neg\psi(s) \wedge T(s, i, s') \Longrightarrow \neg\psi(s') \end{array} \tag{2}$$

The base case states that the initial state must be part of the invariant, and the step case ensures that all states are transitively reachable through the transition relation are also in the invariant. By negating each of the above formulae we obtain an equivalent condition: $\neg\psi$ is an invariant if the two following formulae are unsatisfiable.

$$\begin{array}{ll} \text{(BC)} & \exists\, s : \phi(s) \wedge \psi(s) \\ \text{(SC)} & \exists\, s, i, s' : \neg\psi(s) \wedge T(s, i, s') \wedge \psi(s') \end{array} \tag{3}$$

Both formulae are satisfiability problems (the existential quantifiers are usually omitted) that can be decided with the help of a SAT (or SMT) solver.

The property of interest is often not inductive, however, and the check above fails. An option is to strengthen the property, e.g., using auxiliary invariants obtained using an abstract interpreter. Furthermore, the criterion above can be generalised to k-induction [SSS00, ES03b, HT08, DHKR11]: The predicate $\neg\psi$ is a k-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 $k-1$):

$$\begin{array}{ll}\text{(BC)} & \phi(s_0) \wedge \left(\bigwedge_{0\leq j<k} \neg\psi(s_j) \wedge T(s_j, i_j, s_{j+1})\right) \wedge \psi(s_k) \\ \text{(SC)} & \left(\bigwedge_{0\leq j\leq k} \neg\psi(s_j) \wedge T(s_j, i_j, s_{j+1})\right) \wedge \psi(s_{k+1})\end{array} \tag{4}$$

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 the base case 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 [Hoo93, SS97, KWSS00]. The question is how to solve a sequence of similar SAT problems while reusing effort spent on solving previous instances. The authors of [Str01, WKS01] identify conditions for the reuse of learnt clauses, but this requires expensive book-keeping, which partially saps the benefit of incrementality. Obviously, incremental SAT solving is easy when the modification to the CNF representation of the problem makes it grow monotonically. This means that if we want to solve a sequence of (increasingly constrained) SAT problems with CNF formulae $\Phi(k)$ for $k \geq 0$ then $\Phi(k)$ must be *growing monotonically* in k, i.e. $\Phi(k+1) = \Phi(k) \wedge \varphi(k)$ for CNF formulae $\varphi(k)$. Removal of clauses from $\Phi(k)$ is trickier, as some of the clauses learnt during the solving process are no longer implied by the new instance, and need to be removed as well. This requires additional solver features like solving *under assumptions* [ES03b], 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. We will see that incremental BMC requires a *non-monotonic* series of formulae. In Sect. 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 [GW14] it was observed that some SMT solvers (like Z3) are not optimised for incremental usage and hence perform worse incrementally than non-incrementally.

The bounded model checker that we are using, CBMC [CKL04], itself implements powerful bitvector decision procedures that use a SAT solver such as MINISAT2 [ES03a] as a backend solver. For SAT solvers, solving under assumptions is the prevalent method, hence we will focus on this technique 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 [ES03b] for finite state machines, incremental BMC can be formulated as a sequence of SAT problems $\Phi(k)$ that we need to solve:

$$\begin{aligned} \Phi(0) &:= \phi(s_0) \wedge (\Psi(0) \vee \alpha_0) \\ &\quad \text{with assumption } \neg\alpha_0 \\ \Phi(k+1) &:= \Phi(k) \wedge T(s_k, i_k, s_{k+1}) \wedge \alpha_k \wedge (\Psi(k+1) \vee \alpha_{k+1}) \\ &\quad \text{with assumption } \neg\alpha_{k+1} \end{aligned} \tag{5}$$

where $\Psi(k)$ is the disjunction $\bigvee_{0\leq j\leq k} \psi(s_j)$ of error states $\psi(s_j)$ to be proved unreachable up to iteration k. This disjunction means that the verification fails if *at least one* of the error states is reachable. Since the number of disjuncts in the disjunction $\bigvee_{0\leq j\leq k} \psi(s_j)$ grows in each iteration, our problem is not monotonic: one has to *remove* $\Psi(k)$ when adding $\Psi(k+1)$ because $\Psi(k)$ subsumes $\Psi(k+1)$. This issue can be solved with the help of *solving under assumptions*. 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.[11]

Symbolic execution. In the case of software analysis, the unfolding scheme (5) results in large formulae and would be highly inefficient. 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 the above formula construction and we can replace T by T_k in (5) and ψ by ψ_k in the definition of $\Psi(k)$. T_k denotes the transition formula obtained by symbolic execution of the k^{th} 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, to reduce 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 [HH01, Tip94].

Slicing is performed relative to $\Psi(k)$. We know that the number of disjuncts $\psi(s_j)$ 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 T_k for each time frame k obtained by symbolic execution is a conjunction $\bigwedge_{\tau\in M} \tau$ of subrelations τ (e.g., formulae corresponding to program instructions). We use M to denote the set of these subrelations τ. The slicing operator $slice$ selects a subset of M. The operator $slice$ is monotonic iff for all sets of subrelations M_1, M_2 the following holds: $M_1 \subseteq M_2 \Longrightarrow slice(M_1) \subseteq slice(M_2)$.

We can then view the conjunction of transition relations for k time frames $\widehat{T}(k) = \bigwedge_{0\leq j\leq k} T_j$ 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 $\widehat{T}^{sliced}(k)$:

$$T_{k+1}^{sliced} = \bigwedge_{\tau\in M'_{k+1}\setminus M'_k} \tau. \tag{6}$$

Monotonicity of the 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_k^{sliced} in (5). It is worth mentioning that T_k^{sliced} also contains the 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. Moreover, it takes into account that conditionals could trivially evaluate to false after constant propagation and thus the corresponding branches are not reachable. Then only those τ in M'_{k+1} are added to the formula that have not been in the slice for the previous time frame, resulting in T_{k+1}^{sliced}.

We give an example in Fig. 4. The middle and right-hand side columns give the instructions that are transformed into sets of subrelations M_1 and M_2 in order to build the transition relation T. Note that these subrelations correspond to the simple program on the left-hand side column. The sets of subrelations M'_1 and M'_2 are obtained from the non-greyed instructions in the middle and right-hand side column, respectively. The incremental slice T_2^{sliced} is built from $M'_2 \setminus M'_1$, which corresponds to the bold instructions in the right-hand side column. Note that this incremental slice takes into account a subrelation (corresponding to instruction y=0) that is in M_1, but not in M'_1.

[11] 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.

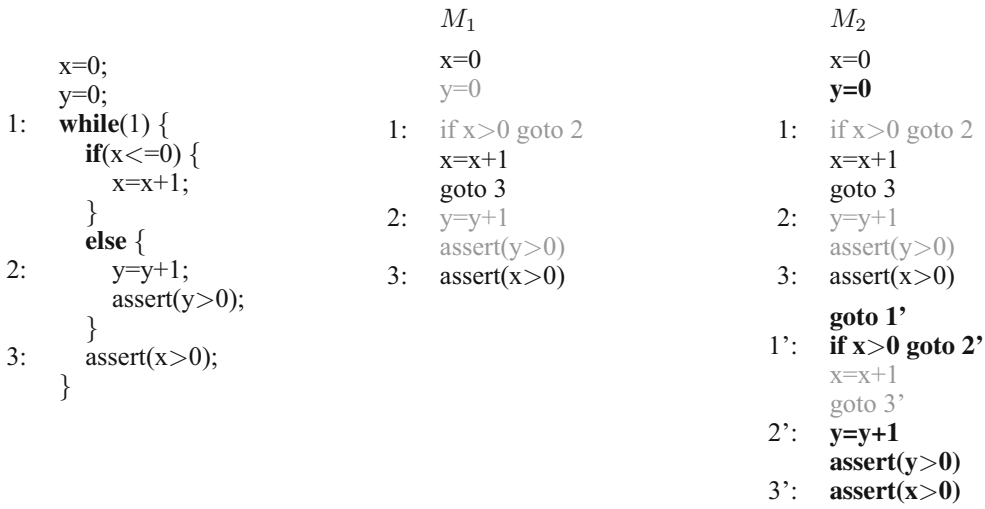


Fig. 4. Incremental slicing

3.3. Incremental refinements

Incremental SAT solving is also used for incremental refinements of the transition relation T for bitvectors and arrays.
Bitvector refinement. The purpose of bitvector refinement [BKO+07, Bie08, HH08, BKO+09, BB09c, EMA10] is to reduce the size of formulae encoding bitvector operations. This is especially important for arithmetic operations that generate huge SAT formulae, e.g. multiplication, division and remainder operations, both for integer and floating-point variables [BKW09]. Bitvector refinement is based on successive under- and over-approximations. For instance, under-approximations can be obtained by fixing a certain number of bits, whereas over-approximation make a certain number of bits unconstrained. If an under-approximation is satisfiable (SAT) or an over-approximation is unsatisfiable (UNSAT) we know that the non-approximated formula is SAT or UNSAT respectively. Otherwise, the number of fixed respectively unconstrained bits is reduced until the non-approximated formula itself is checked.
Arrays. To handle programs with arrays, Ackermann expansion is necessary to ensure the functional consistency property of arrays: $\forall\, i, j : i = j \implies A[i] = A[j]$. However, adding a quadratic number of constraints (in the size of the array A) is extremely costly. Experience has shown that only a small number of these constraints is actually used [PS06].

Hence, it is more efficient trying to solve the SAT formula without these constraints, which is an over-approximation. Hence, if we get an UNSAT result (a), we know that the solution with the Ackermann constraints would be UNSAT, too. In case of a SAT result (b), we check the consistency of the obtained model: if it turns out not to violate consistency, then we know that we have found a real bug. Otherwise (c), we add the violated Ackermann constraint to the formula. The formula construction is trivially monotonic and we can use incremental SAT solving. We repeat the procedure until we hit case (a) or (b), which is guaranteed to happen. Some SMT solvers, such as BOOLECTOR, implement a similar procedure to decide the SMT-LIB array theory [BB09a, BB09b].
Formula construction. Applying above refinements inside an incremental Bounded Model Checker requires using several incremental formula encodings for (in general, non-monotonic) refinements simultaneously. These refinements are global over all unwindings, so that in iteration k we have to further refine transition relations $T_{k'}$ from earlier iterations $k' < k$. We can formalise the incremental formula construction as follows: For iteration $k \geq 0$ of incremental BMC and the ℓ^{th} refinement:

$$\begin{aligned}
\Phi(0,0) &:= \phi(s_0) \wedge (\Psi_0(s_0) \vee \alpha_0) \\
&\quad \text{with assumption } \neg\alpha_0 \\
\Phi(k+1,\ell) &:= \Phi(k,\ell) \wedge (\Psi_{k+1}(s_{k+1}) \vee \alpha_{k+1}) \wedge \alpha_k \wedge \\
&\quad \left(T'_{k+1,\ell}(s_k, i_k, s_{k+1}) \vee \beta_\ell\right) \\
&\quad \text{with assumptions } \neg\alpha_{k+1} \text{ and } \neg\beta_\ell \\
\Phi(k,\ell+1) &:= \Phi(k,\ell) \wedge \left(T'_{k-1,\ell+1}(s_{k-1}, i_{k-1}, s_k) \vee \beta_{\ell+1}\right) \wedge \beta_\ell \\
&\quad \text{with assumptions } \neg\alpha_k \text{ and } \neg\beta_{\ell+1} \\
&\quad \text{for } k \geq 1
\end{aligned} \tag{7}$$

The counter ℓ is incremented in each iteration of the refinement loop until convergence, whereas k is incremented when considering the next time frame. The formulas α_k are the assumptions for the incremental extension of the time frames, whereas the formulas β_ℓ are the assumptions for the refinement iterations.

4. Experimental evaluation

We present the results of our experimental evaluation of incremental BMC and incremental k-induction on industrial programs mainly from the automotive industry. The goal of this evaluation is to quantify the benefit from an incremental approach in a BMC-based tool infrastructure.[12] The experiments described in Sects. 4.2, 4.2 and 4.4 were performed on a 3.5 GHz Intel Xeon machine with 32 GB of physical memory running Windows 7 with a time limit of 3600 s. The evaluation on programs with multiple loops (Sect. 4.5) was run on the StarExec [SST14] cluster infrastructure on 2.40 GHz Intel Xeons running Red Hat Enterprise Linux Workstation release 6.3 (Santiago) with a timeout of 1800 s and a memory limit of 32 GB.

4.1. Implementation

CBMC. We implement our extension [13] for incremental BMC in the Bounded Model Checker for ANSI-C programs CBMC [CKL04] using the SAT solver MINISAT2 [ES03a]. CBMC is called in incremental mode using the command line `cbmc file.c --incremental`. There is an optimised option for programs with a single unbounded loop (see Sect. 4.2). The following options can be added to enable specific features of CBMC:

- `--no-sat-preprocessor`: turns off SAT formula preprocessing, i.e. the MINISAT2 simplifier is not used.
- `--slice-formula`: slices the SAT formula.
- `--refine`: enables bitvector refinement.
- `--unwind-max` k: limits the unwindings of the loop to be checked incrementally to k unwindings. Without this option, CBMC will not terminate for unsatisfiable instances, i.e. bug-free programs with unbounded loops.

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 perform an incremental build of the SAT formula. More information regarding the usage of incremental CBMC can be found on the CPROVER wiki page[14].

Integration with an industrial-strength embedded verification tool. 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 to the master process. Otherwise, i.e. in case of a timeout, the CBMC process is terminated but information about the solved unwindings of the reactive main loop is returned, 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 Sect. 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. Eq. (4) (BC)); in the file for the step case, the variables modified in the loop are havocked, i.e., they are assigned a nondeterministic value at the beginning of the loop. Then the invariant property is assumed, and at the end of the loop the invariant property is asserted (cf. Eq. (4) (SC)). By default, CBMC stops when a counterexample for a property is found, but to check the step case, we require a reversed termination behaviour of CBMC, (option `--stop-when-unsat`), i.e. CBMC continues unwinding as long as the problem is SAT and stops as soon as it is UNSAT.

Implementation of Incremental BMC for General Sequential Programs. Incremental CBMC can also be used for programs with multiple loops. For these programs, CBMC incrementally unwinds loops one at each time. For each loop, the incremental procedure is similar to the one described in Sect. 3.2 for a single unbounded loop. For programs with multiple loops, CBMC will unwind each loop until it is fully unwound or until a maximum depth k is reached. We can detect that a loop is fully unwound at unwinding $j < k$ if all states reached at unwinding j do not satisfy the loop condition. After a loop has been unwound, CBMC continues to the next loop. This procedure is repeated until all loops have been unwound or a bug has been found. Recursive function calls are treated similarly.

Consider the control flow graph (CFG) in Fig. 5(a). The unwinding strategy is illustrated for this CFG in Fig. 5(b). The program has three loops with loop heads 1, 2 and 6 (2 is nested inside 1). The symbolic execution that generates the incremental BMC formula $\Phi(k)$ (see Sect. 3.2) traverses the CFG and stops each time when it encounters an edge in the CFG that returns to a loop head (a so-called *back-edge*). Fig. 5(b) shows three snapshots of the partially unwound CFG that correspond to the parts of the program considered by instances of the incremental BMC formula $\Phi(k)$ for $k = 1, 2, m$. We write $\Phi(1)$ for the formula up to the first back-edge encountered that returns to the loop head of the inner loop (2). Formula $\Phi(2)$ extends $\Phi(1)$ by one further unwinding of the inner loop.

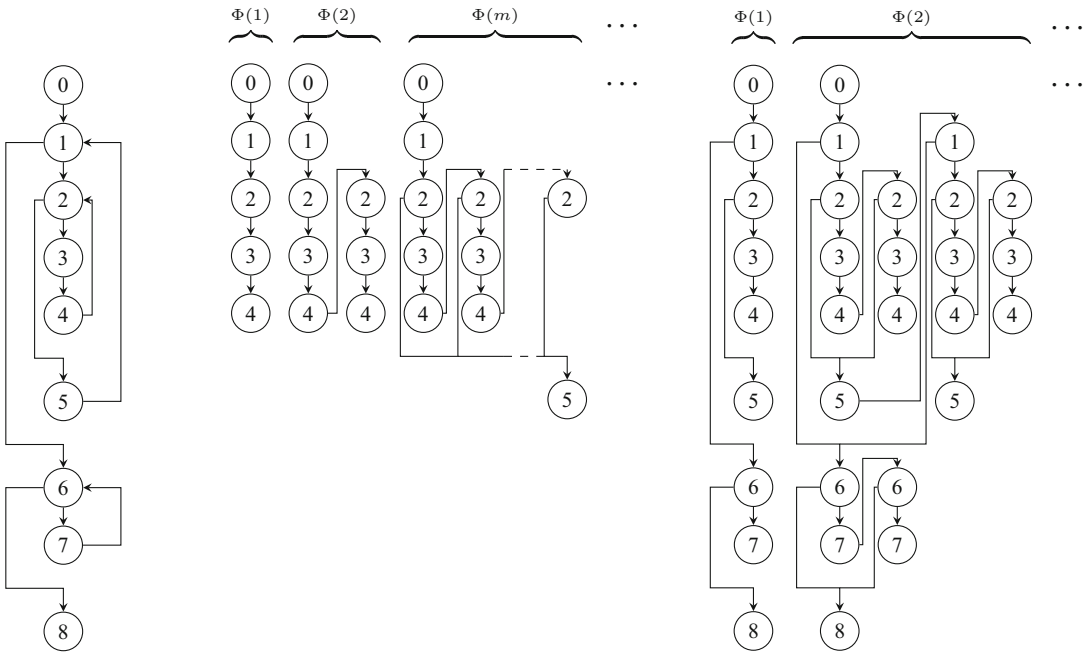
[12] For a comparison with alternative verification approaches, we refer to the results of the Software Verification Competition (http://sv-comp.sosy-lab.org), where BMC-based tools rank in the top 3 every year.

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

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

Table 1. Benchmark characteristics of industrial programs

	LOC	Operators			Input variables			State variables			Observer	
		cond	mul	div/rem	bool	int	float	bool	int	float	bool	Unwindings
SAT												
max	31,222	17,103	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	23,014	49,530	567	37,467	212	282	188	708	663	32	22	10
average	4854	6014	160	1257	30	51	9	163	73	3	7	10



(a) Example program with three loops **(b)** Incremental unwinding strategy of CBMC **(c)** Incremental unwinding strategy of 2LS

Fig. 5. Incremental unwinding strategies

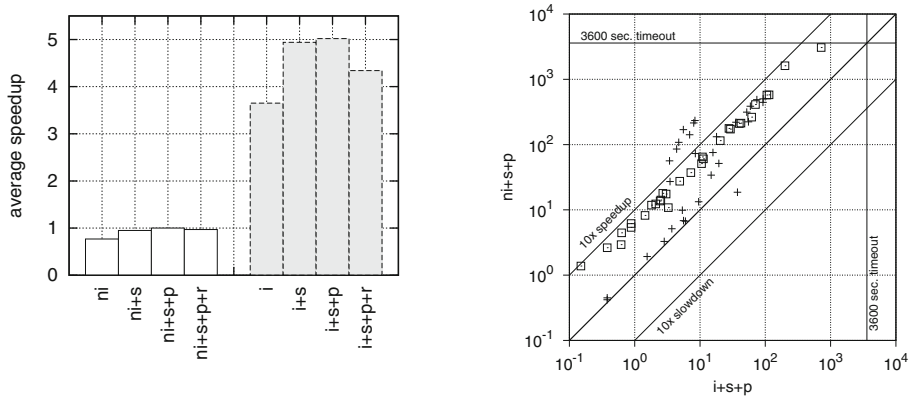
Assume that m is the maximum number of unwindings of the inner loop, then $\Phi(m)$ shows the extension of the formula to the case where the inner loop has been unwound up to this maximum number within the first iteration of the outer loop (with loop head 1). Formula $\Phi(m+1)$ will then extend $\Phi(m)$ by a first unwinding of the inner loop (up to program location 4) for the second iteration of the outer loop. This process continues until a failed assertion or the end of the program (8) is reached.

Implementation of Incremental BMC in 2LS. We implement a different approach to incremental BMC in the static analysis tool 2LS [SK16, BJKS15].[15] 2LS unwinds *all* loops k times and incrementally adds the $(k+1)$th for *all* loops instead of unwinding only the first loop encountered until it has been fully unwound.

We illustrate this unwinding strategy in Fig. 5(c), which shows the first two partial unwindings of the CFG in Fig. 5(a) that correspond to $\Phi(1)$ and $\Phi(2)$, respectively. Formula $\Phi(1)$ consists of one unwinding (up to, but not including the back-edge) for the loops 1, 2, and 6. Formula $\Phi(2)$ then adds another unwinding to each loop. Note that we have two times two unwindings of the inner loop (with loop head 2) now, two for each unwinding of the outer loop (loop head 1).

Structurally, this unwinding strategy is the same as the one that we use in non-incremental CBMC when calling with fixed values for k. In comparison with incrementally unwinding a single loop, the incremental extension of the formula from k to $k+1$ unwindings in Eq. (5) is now also non-monotonic because of T (and not only because of Ψ). This renders many optimisations that non-incremental CBMC performs during symbolic execution such as constant propagation impossible.

[15] Both CBMC and 2LS are built on top of the CPROVER framework. 2LS is publicly available at http://www.cprover.org/2LS.



(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. 6. Incremental versus non-incremental BMC

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.[16] These benchmarks have exactly one unbounded loop. Half of the benchmarks are bug-free (UNSAT instances), half contain a bug (SAT instances). This benchmark suite is suitable for evaluating the 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 — for instance, 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.

For these benchmarks, CBMC is called in incremental mode by using the option `--incremental-check main.0` where `main.0` is the loop identifier of the unbounded loop to be unwound and checked incrementally. The loop identifiers can be obtained using the option `--show-loops`.

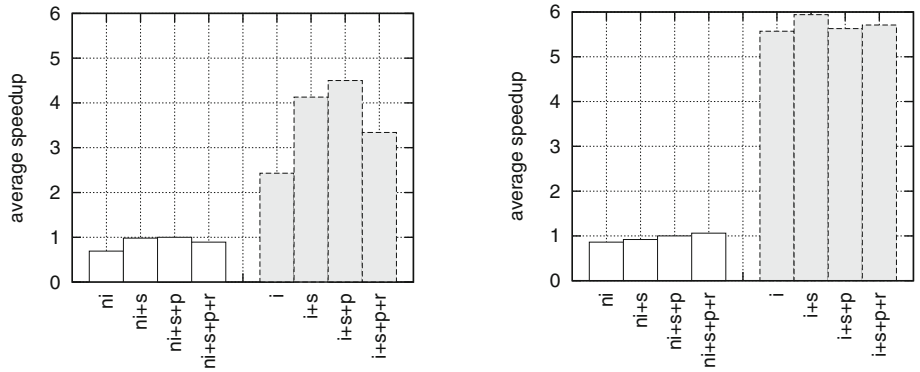
Runtimes. We compared the incremental (i) with the non-incremental (ni) approach and evaluated the impact of slicing (s), SAT preprocessing (p) and bitvector refinement (r).[17] The incremental and non-incremental approaches were compared by activating none of the three techniques, with slicing only (+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. 6a gives the average geometric mean [FW86] speedup of instances that were solved by all approaches. Fig. 7 provides these results split into SAT and UNSAT instances. We consider the (ni+s+p) approach as the baseline since it is the best non-incremental approach. Each bar gives 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× as fast as (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× faster than (ni+s+p) and (i+s+p) is on average over 5× 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), although the effect is less significant for UNSAT than for SAT instances; (ii) not using formula preprocessing is a bad idea in general; and (iii) bitvector refinement provides benefits for UNSAT instances, but produces the overhead for SAT instances, which deteriorates the overall performance of the tool (see Fig. 7(a)). Even though the tool options have some positive effects, they are minor in comparison to the performance gains from using the 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 for the results described in the remainder of the article.

[16] To mitigate this problem, we present a detailed summary of the benchmark characteristics in Table 2 in the "Appendix A".

[17] Array refinement is not used because the benchmarks do not contain arrays.



(a) SAT benchmarks **(b)** UNSAT benchmarks

Fig. 7. Effect of slicing, SAT formula preprocessing and bitvector refinement

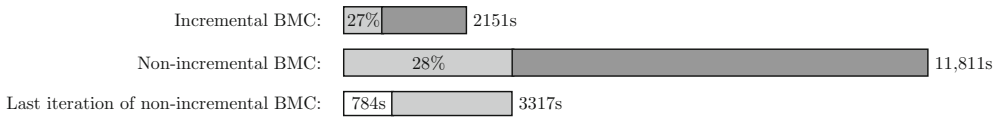


Fig. 8. Solving time versus overall runtime

Fig. 6b is 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 EMBEDDEDTESTER 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: 27% of the time taken by the incremental approach are spent in solving the SAT formula (582 out of 2151 s), compared with 28% of the time taken by the non-incremental approach (3317 out of 11,811 s). We illustrate this observation in the bar chart in Fig. 8, which plots the total runtime consisting of the time spent in generating the SAT formula and solving it (light grey) and the overhead (dark grey) for incremental and non-incremental BMC. Unsurprisingly, as shown in the third bar in Fig. 8, solving the instance for the largest k in the non-incremental approach (white) takes a considerable amount of time (around 24%), when compared to the total time (white+grey) for solving the SAT formulae for iterations 1 to k (784 out of 3317 s).

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 1367k 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 217 and 746k respectively.

Peak memory consumption. 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 significant 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. BTC EMBEDDEDTESTER performs program transformations to improve performance and generates program slices for each test goal. Each of these slices is then passed to CBMC as a subtask. In total, there are 251 subtasks. 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× faster, and for unwinding depth 50 from 377.4 to 108.5 minutes, i.e., more than 3× faster. In the latter case, the rate of solved subproblems for MC/DC (i.e., not run into timeout) could be increased from 98.4% to 99.2%, i.e., two more goals are covered.

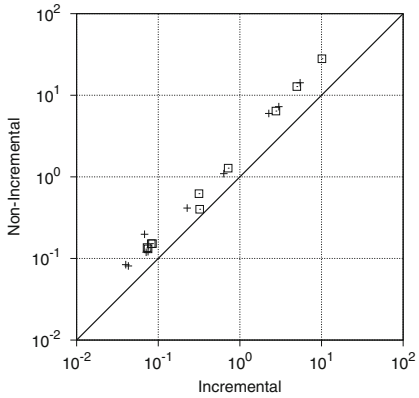


Fig. 9. Incremental k-induction (+ BC instances; □ SC instances)

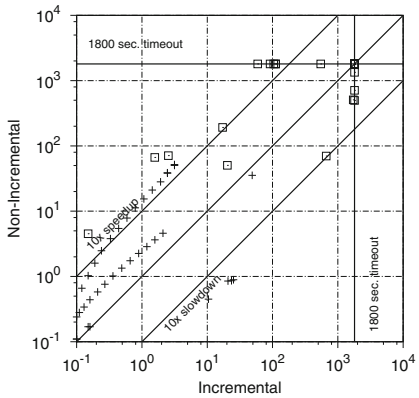


Fig. 10. Incremental versus non-incremental BMC on the SystemC category (+ SAT instances; □ UNSAT instances)

4.4. Incremental k-induction for embedded software

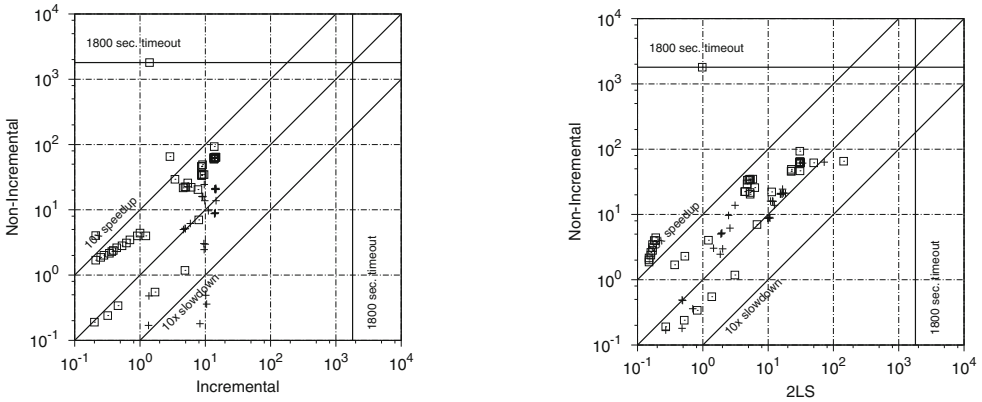
To compare the performance of incremental and non-incremental approaches for k-induction, we considered the subset of UNSAT benchmarks for which k-induction required more than 1 iteration. Note that when k-induction requires only 1 iteration, the performance of both approaches is similar.

Figure 9 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 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× faster than non-incremental checking, on both base and step cases.

4.5. Incremental BMC for programs with multiple loops

Incremental BMC is not restricted to programs with a single loop and may also be applied to programs with multiple loops. To evaluate the performance of incremental BMC on this kind of program, we compared the performance of incremental and non-incremental approaches on the 62 benchmarks from the SystemC category of the Software Verification Competition benchmark set,[18] because these benchmarks, which were derived from SystemC models [CMNR10], contain many loops. Of these benchmarks, 25 are bug-free (UNSAT instances) and 37 contain a bug (SAT instances). These benchmarks have between 2 and 19 loops with an average of 10.3 loops per instance. For SAT instances, the depth of the bug ranges from 1 to 5 with an average depth of 2.5. When compared to industrial benchmarks, SystemC benchmarks are smaller and have shallow bugs, which illustrates some of the differences between industrial and academic benchmarks. For more details on these benchmarks see Table 3 in the "Appendix B".

[18] Available at https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp15.



(a) Incremental vs. non-incremental BMC on additional loop benchmarks (+ SAT instances; □ UNSAT instances)

(b) 2LS vs. non-incremental BMC on additional loop benchmarks (+ SAT instances; □ UNSAT instances)

Fig. 11. Incremental and 2LS versus non-incremental BMC on additional loop benchmarks

We have fixed the maximum number of loop unwindings to 10 for both SAT and UNSAT instances. Note that this unwind depth is larger than the depth of the bugs for the SAT instances. Formula slicing is not yet fully supported in incremental CBMC for programs with multiple loops, and has been disabled for the incremental approach.

Fig. 10 gives a scatter plot with the runtimes of the incremental and non-incremental approaches for SystemC benchmarks. For the majority of the instances, the incremental approach outperforms the non-incremental approach and for many SAT and UNSAT instances the speedup is larger than a factor of 10. However, there are a few instances for which the non-incremental approach performs better. The non-incremental approach unwinds all loops until a fixed unwind depth, whereas the incremental approach fully unwinds one loop before continuing to the next loop. For some instances, fully unwinding each loop may result in the generation of larger formulae, particularly for SAT instances. Not using slicing for the incremental approach may also result in larger formulae. The increase in formula size may explain the observed slowdown for some instances. Overall, when considering instances solved by both approaches, the incremental approach is faster than the non-incremental approach and the average geometric speedup is larger than a factor of 3.

Comparison with 2LS. We compared the incremental BMC implementations of CBMC and 2LS with non-incremental CBMC on 83 benchmarks from the Software Verification Competition benchmark set (categories Simple and Control Flow). These benchmarks are representative for general, i.e. transformational rather than reactive, programs. Most of these programs have only one loop, but the assertion is outside the loop, which distinguishes them from the embedded benchmarks. For more details on these benchmarks see Table 4 in the "Appendix C".

Figure 11 presents the results. Although incremental CBMC is an order of magnitude faster than non-incremental CBMC on many benchmarks, there is a number of SAT benchmarks on which incremental CBMC is significantly slower than the non-incremental version (Fig. 11a). The reason for this is that the unwinding strategy implemented in incremental CBMC is optimised for embedded software with a single unbounded loop (and with the assertions inside the loop). By contrast, this behaviour cannot be observed when comparing 2LS with non-incremental CBMC (Fig. 11b). Although 2LS is slower than incremental CBMC on many benchmarks, the unwinding strategy of 2LS is advantageous for benchmarks where bugs "after" loops can be found with low numbers of unwinding. On such benchmarks 2LS clearly outperforms incremental CBMC and non-incremental CBMC.

We illustrate this observed behavioural difference on the example in Fig. 5(a). Let us assume that the assertion is at program location 3 and that it fails in the third iteration of the inner loop of the first iteration of the outer loop. In this case incremental CBMC can find the bug by only unwinding a very small part of the program that considers only one unwinding of loop 1 and three unwindings of loop 2. By contrast, 2LS constructs a formula that has three unwindings of each loop (and actually nine instances of the inner loop!), which results in a large formula that slows down 2LS in comparison with incremental CBMC. On the other hand, let us assume that the assertion is at program location 8 and that it fails without entering any of the loops. Then the unwinding strategy of 2LS can find the bug in formula $\Phi(1)$, whereas the unwinding strategy of incremental CBMC first has to unwind all the loops up to their maximum number of iterations before it is able to reach location 8.

5. Related work

Most related is recent work on a prototype tool NBIS [GW14], which implements 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 <1 s).[19]

[19] Unfortunately, a working version of the tool was not available.

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 [HRB13, MMBC11], test case generation [PRS+12], symbolic analysis for improving simulation coverage [AKRS08], and directed random testing [SYR08]. Yet, all these works have not exploited incremental BMC.

The test vector generation tool FSHELL [HSTV09] 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 [CDE08] 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.

Incremental SAT solving has important applications in other verification techniques like the IC3 algorithm [Bra12, EMB11] and incremental BMC is standard for hardware verification [JS05, Wie11]. We show that the speedups of incremental SAT solving reported in [ES03b] regarding k-induction on small HW circuits carry over to industrial embedded software.

6. Conclusions and future work

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. We implement support for incremental BMC for programs with *multiple loops* in two tools, using different loop unwinding strategies. Our experimental evaluation shows that the version of incremental BMC implemented in CBMC works well on programs with multiple loops that are akin to embedded programs, whereas 2LS's approach is better suited for general programs. Even though the engineering aspects of both approaches for multiple loops can still be improved, we already observe significant speedups in comparison to the non-incremental approach that show the applicability of incremental BMC beyond embedded software.

There are several opportunities to further improve the performance of BMC and k-induction for embedded programs in practice. It is often difficult to find bugs that require many unwindings using BMC because of the exponentially increasing amount of time and memory necessary to solve the generated SAT formulae. Kroening et al. [KLW15] present a *loop acceleration* technique that is sound for BMC, i.e. it adds short-cut paths to the program that have the effect of many loop iterations without introducing spurious behaviour. We would like to investigate how this technique can be combined with incremental BMC.

It has been shown [BDW15, BJKS15] that powerful verification tools can be built by strengthening the step case in k-induction with additional invariants that are inferred using abstract interpretation techniques. This approach can be further extended by using incremental loop unwindings [BJKS15]. However, a comprehensive study on the practical benefit for embedded programs has not yet been conducted. Regarding general programs, we are planning to implement support for recursion in 2LS so that we can compare it with incremental unfolding of recursions in CBMC. Also, we would like to add a slicing operator that supports multiple loops and recursion.

A promising application of incremental BMC is the analysis of concurrent programs through sequentialisations (e.g. [ITF+14]). Incrementality could be exploited in two ways in this context: by incrementally increasing the number of unwinding of loops (which might also augment the number of threads) and for increasing the number of context switches that are considered. The challenge is to find good encodings of these sequentialisations that allow us to use incremental SAT solving efficiently.

Appendix A: Industrial benchmark characteristics

See Table 2.

Appendix B: SystemC benchmark characteristics

See Table 3.

Appendix C: Additional loop benchmarks characteristics

See Table 4.

Table 2. Embedded software benchmark characteristics (name of the benchmark and application domain, lines of code, number of operators (cond(a?b:c), mul(*), div/rem(/,%)), number of boolean/integer/floating point input and state variables, number of boolean variables introduced by the observer instrumentation, number of loop unwindings considered; k-induction was performed on the instances marked with *)

Name	LOC	Operators			Input variables			State variables			Observer	
		cond	mul	div/rem	bool	int	float	bool	int	float	bool	Unwindings
automotive_sat_01	3762	2032	82	1	14	282	0	229	50	0	3	12
automotive_sat_02	1854	189	79	1	78	4	0	165	7	0	3	15
automotive_sat_03	15,277	17,103	669	75	230	244	0	868	275	0	1	9
automotive_sat_04	13,853	16,908	601	59	208	219	0	741	266	0	1	12
automotive_sat_05	469	193	90	11	1	0	0	17	3	0	3	21
automotive_sat_06	10,702	5117	646	1	7	54	19	28	60	22	16	5
automotive_sat_07	10,970	5068	646	1	7	54	19	27	62	22	15	4
automotive_sat_08	3656	2657	79	1	14	61	26	20	68	30	16	2
automotive_sat_09	253	34	79	1	0	3	0	23	4	0	3	103
automotive_sat_10	604	117	79	1	23	7	0	81	10	0	3	40
automotive_sat_11	592	115	79	1	23	7	0	79	10	0	3	48
automotive_sat_12	1978	2201	79	1	0	0	0	4	172	0	3	53
automotive_sat_13	1980	2198	79	1	0	0	0	4	172	0	3	55
automotive_sat_14	1222	216	79	1	0	26	0	94	67	0	3	56
automotive_sat_15	5020	3172	79	1	18	4	0	115	22	0	3	17
automotive_sat_16	2578	4572	89	4	1	20	105	3	22	107	17	2
automotive_sat_17	2580	4592	89	4	1	20	105	2	22	107	18	1
automotive_sat_18	2740	4718	89	4	1	20	105	2	24	107	16	2
automotive_sat_19	27,456	3579	177	7	546	95	0	3426	438	0	1	12
automotive_sat_20	27,456	3579	177	7	546	95	0	3426	438	0	1	16
automotive_sat_21	31,222	3705	178	7	688	477	0	3876	750	0	1	12
automotive_sat_22	30,834	3620	177	7	652	476	0	3837	744	0	1	14
automotive_sat_23	1270	508	102	5	6	66	0	79	124	9	16	1
automotive_sat_24	1272	501	102	5	6	66	0	78	124	9	17	3
automotive_sat_25	1282	506	102	5	6	67	0	79	128	9	15	1
automotive_sat_26	321	28	79	1	6	2	0	36	2	0	3	106
avionics_sat	2214	1413	79	2	30	16	0	189	52	0	1	20
fuelsys_sat_01	9402	16,603	311	6	0	0	4	31	5	8	22	1
fuelsys_sat_02	9404	16,757	311	6	0	0	4	31	5	8	22	1
fuelsys_sat_03	5746	8521	224	3	0	0	4	30	5	7	19	1
automotive_unsat_01*	3761	2032	82	1	14	282	0	229	50	0	3	10
automotive_unsat_02	3762	2032	82	1	14	282	0	229	50	0	3	10
automotive_unsat_03	1579	889	79	1	0	38	0	75	4	0	3	10
automotive_unsat_04	1853	189	79	1	78	4	0	165	7	0	3	10
automotive_unsat_05	503	321	106	19	1	0	0	21	3	0	3	10
automotive_unsat_06	13,259	16,672	545	59	188	207	0	708	232	0	1	10
automotive_unsat_07	464	193	90	11	1	0	0	17	3	0	3	10
automotive_unsat_08	23,014	49,530	536	37,467	92	220	0	697	304	0	1	10
automotive_unsat_09	4768	3334	79	1	0	26	0	215	663	0	3	10
automotive_unsat_10	1035	160	79	1	30	4	0	115	29	0	1	10
automotive_unsat_11	12142	5859	567	0	7	54	19	27	60	22	17	10
automotive_unsat_12	12518	6242	567	0	7	54	19	27	62	22	15	10
automotive_unsat_13*	4726	3091	42	0	14	61	26	30	71	32	16	10
automotive_unsat_14*	591	115	79	1	23	7	0	79	10	0	3	10
automotive_unsat_15*	1977	2198	79	1	0	0	0	4	172	0	3	10
automotive_unsat_16	2339	559	82	9	22	56	0	170	79	0	3	10
automotive_unsat_17*	1399	258	79	1	0	29	0	106	73	0	3	10
automotive_unsat_18*	5021	3172	79	1	18	4	0	115	22	0	3	10
automotive_unsat_19*	7979	12,127	119	15	0	0	0	5	16	0	3	10
automotive_unsat_20*	6217	686	88	2	212	87	0	697	60	0	1	10
automotive_unsat_21*	5230	1043	81	2	99	24	0	511	112	0	1	10
automotive_unsat_22	190	97	90	11	0	0	0	4	31	0	1	10
automotive_unsat_23	659	93	79	1	9	1	0	75	10	0	3	10
automotive_unsat_24	3554	787	81	52	16	79	0	226	45	0	3	10
automotive_unsat_25	1575	184	79	1	38	0	0	199	15	0	3	10
avionics_unsat	2329	1413	79	2	30	16	0	188	52	0	1	10
fuelsys_unsat_01*	5146	17,271	214	5	0	0	3	11	0	5	21	10
fuelsys_unsat_02	7806	19,764	215	6	0	0	4	31	5	8	22	10
fuelsys_unsat_03	7804	19,764	215	5	0	0	4	31	5	8	22	10
fuelsys_unsat_04	3340	11,671	205	3	0	0	3	15	0	3	18	10

Table 3. SystemC benchmark characteristics (name of the benchmark, lines of code, number of loops, and number of loop unwindings considered)

Name	LOC	Loops	Unwindings
bist_cell_unsat.cil.c	240	2	10
kundu_unsat.cil.c	290	5	10
mem_slave_tlm.1_unsat.cil.c	724	13	10
mem_slave_tlm.2_unsat.cil.c	729	13	10
mem_slave_tlm.3_unsat.cil.c	734	13	10
mem_slave_tlm.4_unsat.cil.c	739	13	10
mem_slave_tlm.5_unsat.cil.c	744	13	10
pc_sfifo_1_unsat.cil.c	172	4	10
pc_sfifo_2_unsat.cil.c	214	4	10
pc_sfifo_3_unsat.cil.c	258	4	10
pipeline_unsat.cil.c	400	3	10
token_ring.01_unsat.cil.c	210	5	10
token_ring.02_unsat.cil.c	270	6	10
token_ring.03_unsat.cil.c	330	7	10
token_ring.04_unsat.cil.c	390	8	10
token_ring.05_unsat.cil.c	450	9	10
token_ring.06_unsat.cil.c	510	10	10
token_ring.07_unsat.cil.c	570	11	10
token_ring.08_unsat.cil.c	630	12	10
token_ring.09_unsat.cil.c	690	13	10
token_ring.10_unsat.cil.c	750	14	10
token_ring.11_unsat.cil.c	810	15	10
token_ring.12_unsat.cil.c	870	16	10
token_ring.13_unsat.cil.c	930	17	10
toy_unsat.cil.c	315	6	10
kundu1_sat.cil.c	233	4	3
kundu2_sat.cil.c	285	5	2
pc_sfifo_1_sat.cil.c	173	4	1
pc_sfifo_2_sat.cil.c	215	4	1
pipeline_sat.cil.c	400	3	5
token_ring.01_sat.cil.c	217	5	3
token_ring.02_sat.cil.c	277	6	3
token_ring.03_sat.cil.c	337	7	3
token_ring.04_sat.cil.c	397	8	3
token_ring.05_sat.cil.c	457	9	3
token_ring.06_sat.cil.c	517	10	3
token_ring.07_sat.cil.c	577	11	3
token_ring.08_sat.cil.c	637	12	3
token_ring.09_sat.cil.c	697	13	3
token_ring.10_sat.cil.c	757	14	3
token_ring.11_sat.cil.c	817	15	3
token_ring.12_sat.cil.c	877	16	3
token_ring.13_sat.cil.c	937	17	3
token_ring.14_sat.cil.c	875	16	3
token_ring.15_sat.cil.c	935	17	3
toy1_sat.cil.c	317	6	3
toy2_sat.cil.c	314	6	3
transmitter.01_sat.cil.c	197	6	2
transmitter.02_sat.cil.c	256	7	2
transmitter.03_sat.cil.c	315	8	2
transmitter.04_sat.cil.c	374	9	2
transmitter.05_sat.cil.c	433	10	2
transmitter.06_sat.cil.c	492	11	2
transmitter.07_sat.cil.c	551	12	2
transmitter.08_sat.cil.c	610	13	2
transmitter.09_sat.cil.c	669	14	2
transmitter.10_sat.cil.c	728	15	2
transmitter.11_sat.cil.c	787	16	2
transmitter.12_sat.cil.c	846	17	2
transmitter.13_sat.cil.c	905	18	2
transmitter.15_sat.cil.c	905	18	1
transmitter.16_sat.cil.c	961	19	1

Table 4. Additional loop benchmark characteristics (name of the benchmark, lines of code, number of loops, and number of loop unwindings considered)

Name	LOC	Loops	Unwindings
cdaudio_unsat.i.cil.c	8433	20	10
diskperf_unsat.i.cil.c	4285	2	10
floppy2_unsat.i.cil.c	30,942	181	10
floppy_unsat.i.cil.c	7772	23	10
parport_unsat.i.cil.c	10271	37	10
s3_clnt.blast.01_unsat.i.cil.c	1585	1	10
s3_clnt.blast.02_unsat.i.cil.c	1583	1	10
s3_clnt.blast.03_unsat.i.cil.c	1583	1	10
s3_clnt.blast.04_unsat.i.cil.c	1583	1	10
s3_srvr.blast.01_unsat.i.cil.c	1686	1	10
s3_srvr.blast.02_unsat.i.cil.c	1682	1	10
s3_srvr.blast.06_unsat.i.cil.c	1750	1	10
s3_srvr.blast.07_unsat.i.cil.c	1702	1	10
s3_srvr.blast.08_unsat.i.cil.c	1706	1	10
s3_srvr.blast.09_unsat.i.cil.c	1702	1	10
s3_srvr.blast.10_unsat.i.cil.c	1694	1	10
s3_srvr.blast.11_unsat.i.cil.c	1702	1	10
s3_srvr.blast.12_unsat.i.cil.c	1714	1	10
s3_srvr.blast.13_unsat.i.cil.c	1702	1	10
s3_srvr.blast.14_unsat.i.cil.c	1726	1	10
s3_srvr.blast.15_unsat.i.cil.c	1718	1	10
s3_srvr.blast.16_unsat.i.cil.c	1738	1	10
cdaudio_simpl1_unsat.cil.c	2899	1	10
diskperf_simpl1_unsat.cil.c	1413	1	10
floppy_simpl3_unsat.cil.c	1467	1	10
floppy_simpl4_unsat.cil.c	2056	1	10
s3_clnt_1_unsat.cil.c	757	1	10
s3_clnt_2_unsat.cil.c	763	1	10
s3_clnt_3_unsat.cil.c	796	1	10
s3_clnt_4_unsat.cil.c	763	1	10
s3_srvr_1_unsat.cil.c	862	1	10
s3_srvr_1a_unsat.cil.c	200	1	10
s3_srvr_1b_unsat.cil.c	130	1	10
s3_srvr_2_unsat.cil.c	849	1	10
s3_srvr_3_unsat.cil.c	848	1	10
s3_srvr_4_unsat.cil.c	849	1	10
s3_srvr_6_unsat.cil.c	943	1	10
s3_srvr_7_unsat.cil.c	874	1	10
s3_srvr_8_unsat.cil.c	884	1	10
test_locks_10_unsat.c	116	1	10
test_locks_11_unsat.c	126	1	10
test_locks_12_unsat.c	136	1	10
test_locks_13_unsat.c	146	1	10
test_locks_14_unsat.c	156	1	10
test_locks_15_unsat.c	166	1	10
test_locks_5_unsat.c	66	1	10
test_locks_6_unsat.c	76	1	10
test_locks_7_unsat.c	86	1	10
test_locks_8_unsat.c	96	1	10
test_locks_9_unsat.c	106	1	10
s3_clnt.blast.01_sat.i.cil.c	1585	1	7
s3_clnt.blast.02_sat.i.cil.c	1583	1	6
s3_clnt.blast.03_sat.i.cil.c	1583	1	6
s3_clnt.blast.04_sat.i.cil.c	1583	1	6
s3_srvr.blast.01_sat.i.cil.c	1686	1	4
s3_srvr.blast.02_sat.i.cil.c	1682	1	4
s3_srvr.blast.03_sat.i.cil.c	1682	1	4
s3_srvr.blast.04_sat.i.cil.c	1682	1	4
s3_srvr.blast.06_sat.i.cil.c	1747	1	6
s3_srvr.blast.07_sat.i.cil.c	1702	1	6
s3_srvr.blast.08_sat.i.cil.c	1703	1	10
s3_srvr.blast.09_sat.i.cil.c	1702	1	6
s3_srvr.blast.10_sat.i.cil.c	1691	1	10

Table 4. continued

Name	LOC	Loops	Unwindings
s3_srvr.blast.11_sat.i.cil.c	1702	1	5
s3_srvr.blast.12_sat.i.cil.c	1711	1	6
s3_srvr.blast.13_sat.i.cil.c	1702	1	6
s3_srvr.blast.14_sat.i.cil.c	1723	1	6
s3_srvr.blast.15_sat.i.cil.c	1715	1	10
s3_srvr.blast.16_sat.i.cil.c	1735	1	6
s3_clnt_1_sat.cil.c	757	1	6
s3_clnt_2_sat.cil.c	763	1	6
s3_clnt_3_sat.cil.c	796	1	6
s3_clnt_4_sat.cil.c	763	1	6
s3_srvr_10_sat.cil.c	872	1	1
s3_srvr_11_sat.cil.c	863	1	7
s3_srvr_12_sat.cil.c	960	1	6
s3_srvr_13_sat.cil.c	885	1	4
s3_srvr_14_sat.cil.c	892	1	2
s3_srvr_1_sat.cil.c	858	1	4
s3_srvr_2_sat.cil.c	849	1	4
s3_srvr_6_sat.cil.c	946	1	1
test_locks_14_sat.c	156	1	1
test_locks_15_sat.c	156	1	1

References

[AKRS08] Alur R, Kanade A, Ramesh S, Shashidhar KC (2008) Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In: International conference on embedded software. ACM, pp 89–98

[BB09a] Brummayer R, Biere A (2009) Boolector: an efficient SMT solver for bit-vectors and arrays. In: Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 174–177

[BB09b] Brummayer R, Biere A (2009) Lemmas on demand for the extensional theory of arrays. J Satisf Boolean Model Comput 6(1–3):165–201

[BB09c] Brummayer R, Biere A (2009) Effective bit-width and under-approximation. In: Computer aided systems theory. Springer, Berlin, pp 304–311

[BCCZ99] Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 193–207

[BDW15] Beyer D, Dangl M, Wendler P (2015) Boosting k-induction with continuously-refined invariants. In Computer-aided verification. Springer, Berlin, pp 622–640

[Bie08] Biere A (2008) PicoSAT essentials. J Satisf Boolean Model Comput 4(2–4):75–97

[BJKS15] Brain M, Joshi S, Kroening D, Schrammel P (2015) Safety verification and refutation by k-invariants and k-induction. In: Static analysis symposium. Springer, Berlin, pp 145–161

[BKO+07] Bryant RE, Kroening D, Ouaknine J, Seshia SA, Strichman O, Brady BA (2007) Deciding bit-vector arithmetic with abstraction. In: Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 358–372

[BKO+09] Bryant RE, Kroening D, Ouaknine J, Seshia SA, Strichman O, Brady BA (2009) An abstraction-based decision procedure for bit-vector arithmetic. J Softw Tools Technol Transf 11(2):95–104

[BKW09] Brillout A, Kroening D, Wahl T (2009) Mixed abstractions for floating-point arithmetic. In: Formal methods in computer-aided design. IEEE, pp 69–76

[Bra12] Bradley AR (2012) IC3 and beyond: incremental, inductive verification. In: Computer-aided verification. Springer, Berlin, p 4

[Bue62] Buechi Julius R (1962) On a decision method in restricted second-order arithmetic. In: International congress on logic, methodology, and philosophy of science. Stanford University Press, Stanford, pp 1–11

[CBRZ01] Clarke E, Biere A, Raimi R, Zhu Y (2001) Bounded model checking using satisfiability solving. Form Methods Syst Des 19(1):7–34

[CDE08] Cadar C, Dunbar D, Engler DR (2008) KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Operating systems design and implementation. USENIX Association, pp 209–224

[CKL04] Clarke EM, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 168–176

[CMNR10] Cimatti A, Micheli A, Narasamdya I, Roveri M (2010) Verifying SystemC: a software model checking approach. In: Form Methods Comput Aided Des. IEEE, pp 51–59

[CRT10] Chakraborty S, Ramesh S, Teich J (2010) Model-based analysis, synthesis and testing of automotive hardware/software architectures. In: International conference on embedded software. ACM, pp 299–300

[NT10] Dias Neto AC, Horta Travassos G (2010) A picture from the model-based testing area: concepts, techniques, and challenges. Adv Comput 80:45–120

[DHKR11] Donaldson A, Haller L, Kroening D, Rümmer P (2011) Software verification using k-induction. In: Static analysis symposium. Springer, Berlin, pp 351–368

[EMA10] Eén N, Mishchenko A, Amla N (2010) A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In: Formal methods in computer-aided design. IEEE, pp 181–188

[EMB11] Eén N, Mishchenko A, Brayton RK (2011) Efficient implementation of property directed reachability. In: Formal Methods in Computer-Aided Design. IEEE, pp 125–134

[ES03a] Eén N, Sörensson N (2003) An extensible SAT-solver. In: Theory and applications of satisfiability testing. Springer, Berlin, pp 502–518
[ES03b] Eén N, Sörensson N (2003) Temporal induction by incremental SAT solving. Electron Notes Theor Comput Sci 89:4:543–560
[FW86] Fleming P, Wallace J (1986) How not to lie with statistics: the correct way to summarize benchmark results. Commun ACM, 29(3):218–221
[FWA09] Fraser G, Wotawa F, Ammann P (2009) Testing with model checkers: a survey. Softw Test Verif Reliab 19(3):215–261
[GKF+12] Gunnarsson D, Kuntz S, Farrall G, Iwai A, Ernst R (2012) Trends in automotive embedded systems. In: International conference on hardware/software codesign and system synthesis. IEEE, pp 9–10
[GW14] Günther H, Weissenbacher G (2014) Incremental bounded software model checking. ACM, pp 40–47
[Hal93] Halbwachs N (1993) Synchronous programming of reactive systems. Kluwer
[HH01] Harman M, Hierons RM (2001) An overview of program slicing. Softw Focus 2(3):85–92
[HH08] He N, Hsiao MS (2008) A new testability guided abstraction to solving bit-vector formula. In: International workshop on bit-precise reasoning
[Hoo93] Hooker JN (1993) Solving the incremental satisfiability problem. J Log Algebraic Program 15(1&2):177–186
[HRB13] Herber P, Reicherdt R, Bittner P (2013) Bit-precise formal verification of discrete-time MATLAB/Simulink models using SMT solving. In: International conference on embedded software, pp 1–10
[HSTV09] Holzer A, Schallhart C, Tautschnig M, Veith H (2009) Query-driven program testing. In: Verification, model checking, and abstract interpretation. Springer, Berlin, pp 151–166
[HT08] Hagen G, Tinelli C (2008) Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Formal methods in computer-aided design. IEEE, pp 1–9
[HVCR01] Hayhurst KJ, Veerhusen DS, Chilenski JJ, Rierson LK (2001) A practical tutorial on modified condition/decision coverage. Technical report, NASA
[ISO11] ISO 26262: Road vehicles—functional safety (2011)
[ITF+14] Inverso O, Tomasco E, Fischer B, La Torre S, Parlato G (2014) Bounded model checking of multi-threaded C programs via lazy sequentialization. In: Computer-aided verification. Springer, Berlin, pp 585–602
[JS05] Jin H, Somenzi F (2005) An incremental algorithm to check satisfiability for bounded model checking. Electron Notes Theor Comput Sci 119:2:51–65
[KLW15] Kroening D, Lewis M, Weissenbacher G (2015) Under-approximating loops in C programs for fast counterexample detection. Form Methods Syst Des 47(1):75–92
[KOS+11] Kroening D, Ouaknine J, Strichman O, Wahl T, Worrell J (2011) Linear completeness thresholds for bounded model checking. In: Computer-aided verification. Springer, pp 557–572
[KS03] Kroening D, Strichman O (2003) Efficient computation of recurrence diameters. In: Verification, model checking, and abstract interpretation. Springer, Berlin, pp 298–309
[KT14] Kroening D, Tautschnig M (2014) CBMC—C bounded model checker—(competition contribution). In: Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 389–391
[KWSS00] Kim J, Whittemore J, Sakallah KA, Marques Silva JP (2000) On applying incremental satisfiability to delay fault testing. In: Design automation and test in Europe. IEEE, pp 380–384
[MMBC11] Manamcheri K, Mitra S, Bak S, Caccamo M (2011) A step towards verification and synthesis from simulink/stateflow models. In: Hybrid systems: computation and control. ACM, pp 317–318
[PdSSM12] Petrenko A, da Silva Simão A, Maldonado JC (2012) Model-based testing of software and systems: recent advances and challenges. J Softw Tools Technol Transf 14(4):383–386
[PRS+12] Peranandam P, Raviram S, Satpathy M, Yeolekar A, Gadkari AA, Ramesh S (2012) An integrated test generation tool for enhanced coverage of simulink/stateflow models. In: Design automation and test in Europe. IEEE, pp 308–311
[PS06] Pnueli A, Strichman O (2006) Reduced functional consistency of uninterpreted functions. Electron Notes Theor Comput Sci 144(2):53–65
[SK16] Schrammel P, Kroening D (2016) 2LS for program analysis—(competition contribution). In: Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 905–907
[SKB+15] Schrammel P, Kroening D, Brain M, Martins R, Teige T, Bienmüller T (2015) Successful use of incremental BMC in the automotive industry. In: Formal methods for industrial critical systems. Springer, Berlin, pp 62–76
[SS97] Silva JM, Sakallah KA (1997) Robust search algorithms for test pattern generation. IEEE, pp 152–161
[SSS00] Sheeran M, Singh S, Stålmarck G (2000) Checking safety properties using induction and a SAT-solver. In: Formal methods in computer-aided design, volume 1954 of LNCS. IEEE, pp 108–125
[SST14] Stump A, Sutcliffe G, Tinelli C (2014) StarExec: a cross-community infrastructure for logic solving. In: International joint conference on automated reasoning, pp 367–373
[Str01] Strichman O (2001) Pruning techniques for the SAT-based bounded model checking problem. Springer, Berlin, pp 58–70
[SYR08] Satpathy M, Yeolekar A, Ramesh S (2008) Randomized directed testing (REDIRECT) for simulink/stateflow models. In: International conference on embedded software, pp 217–226
[Tip94] Tip F (1994) A survey of program slicing techniques. Technical report, CWI-Amsterdam
[Wie11] Wieringa S (2011) On incremental satisfiability and bounded model checking. In: Design and implementation of formal tools and systems, pp 46–54
[WKS01] Whittemore J, Kim J, Sakallah KA (2001) SATIRE: a new incremental satisfiability engine. In: Design automation conference. ACM, pp 542–545

Received 1 May 2016
Accepted in revised form 16 December 2016 by Michael Butler and Jim Woodcock
Published online 22 February 2017