Incremental Bounded Software Model Checking

Henning Günther
Vienna University of Technology, Austria

Georg Weissenbacher
Vienna University of Technology, Austria

ABSTRACT

Conventional Bounded Software Model Checking tools generate a symbolic representation of all feasible executions of a program up to a predetermined bound. An insufficiently large bound results in missed bugs, and a subsequent increase of the bound necessitates the complete reconstruction of the instance and a restart of the underlying solver. Conversely, exceedingly large bounds result in prohibitively large decision problems, causing the verifier to run out of resources before it can provide a result.

We present an incremental approach to Bounded Software Model Checking, which enables increasing the bound without incurring the overhead of a restart. Further, we provide an LLVM-based open-source implementation which supports a wide range of incremental SMT solvers. We compare our implementation to other traditional non-incremental software model checkers and show the advantages of performing incremental verification by analyzing the overhead incurred on a common suite of benchmarks.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Bounded Model Checking*

General Terms

Experimentation, Performance

Keywords

LLVM, Incremental, SMT, C

1. INTRODUCTION

Bounded Model Checking (BMC) is arguably one of the most successful and widely used formal verification techniques, as witnessed by the TACAS most influential paper award for Biere et al.'s seminal paper [5]. As BMC performs a symbolic exploration of execution traces up to a bounded length only, the primary application of the technique is the detection of bugs. While BMC was initially aimed at hardware designs, it has since become a standard technique for software verification that is implemented in numerous verification tools [13].

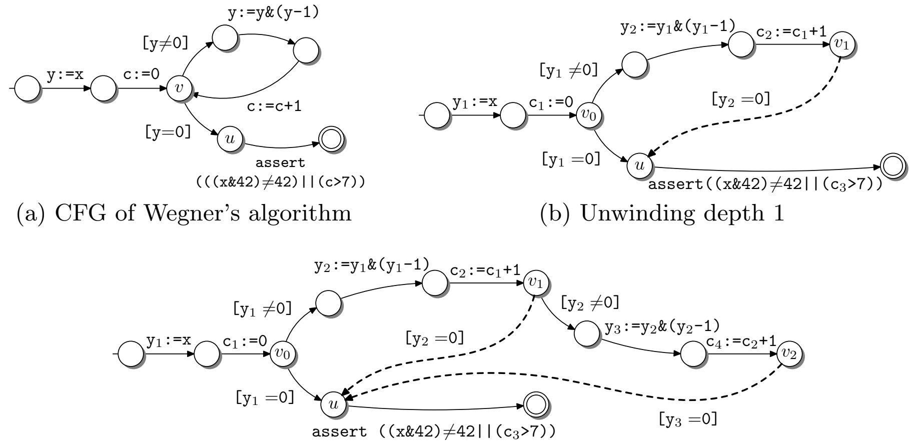
We illustrate the inner workings of a typical implementation of BMC for software using the program in Figure 1a. The program implements Wegner's algorithm [25] to assert that more than 7 bits (or flags) in a bit-vector `x` are set if `x` matches a certain bit-mask. Bounded software model checking tools such as LLBMC [21] or CBMC [9] unwind the control flow graph (CFG) of the program into a directed acyclic graph (DAG) until a certain user-specified bound is reached, and convert the resulting loop-free code into static single assignment (SSA) form [11]. Figures 1b and 1c illustrate this process for the unwinding depths one and two, respectively. To avoid a blowup of the DAG, the loop exit edges (dashed in Figure 1) are *merged* after each loop iteration. Since each variable is assigned only once along each path in SSA form, this requires a case split to determine the value of variable `c` at node u. (In the SSA representation, this is typically indicated using a ϕ function $\mathtt{c}_3 := \phi(\mathtt{c}_1, \mathtt{c}_2)$.) For Figure 1b, we obtain the encoding in Figure 2.

By negating the assertion we achieve that any satisfying assignment (provided by a satisfiability checker) of this instance corresponds to a program execution that violates the assertion. The given instance, however, is unsatisfiable, indicating that there is no path that traverses the loop at most one time and violates the assertion. To detect a bug, we need to increment the loop bound and *reconstruct* the formula (since the disjunctive case split cannot be augmented), or provide a sufficiently large bound in the first place.

For the given program, determining that 3 is the smallest bound admitting an assertion violation is non-trivial unless one understands that the assignment `y:=y&(y-1)` resets the right-most bit in `y` that is one. Safe over-estimations (e.g., the bit-width of `x`) lead to unnecessarily hard problem instances for the satisfiability solver, and under-estimations necessitate expensive restarts.

BMC tools deploy contemporary SAT or SMT solvers and benefit greatly from the impressive advances in this field [6]. A characteristic of most modern SMT solvers is that they solve formulas *incrementally*, reusing the results of previous calls whenever the formula is augmented with additional conjuncts. Additionally, incremental solvers make it possible to add formulas on a tentative basis and later retract them without requiring a restart.

SPIN '14, July 21–23, 2014, San Jose, CA, USA



(a) CFG of Wegner's algorithm

(b) Unwinding depth 1

(c) Unwinding depth 2

Figure 1: Incremental unwinding of a control flow graph

$$
\begin{array}{lr}
(\mathtt{y}_1 = \mathtt{x}) \wedge (\mathtt{c}_1 = 0) \wedge (\mathtt{c}_2 = \mathtt{c}_1 + 1) \wedge (\mathtt{y}_2 = \mathtt{y}_1 \& (\mathtt{y}_1 - 1)) & \text{(assignments)} \\
((\mathtt{y}_1 = 0) \wedge (\mathtt{c}_3 = \mathtt{c}_1)) \vee ((\mathtt{y}_1 \neq 0) \wedge (\mathtt{y}_2 = 0) \wedge (\mathtt{c}_3 = \mathtt{c}_2)) & \text{(case split)} \\
((\mathtt{y}_1 = 0) \vee ((\mathtt{y}_1 \neq 0) \wedge (\mathtt{y}_2 = 0))) \wedge ((\mathtt{x}\&42 = 42) \wedge (\mathtt{c}_3 \leq 7)) & \text{(negated assertion)}
\end{array}
$$

Figure 2: Encoding of unwinding in Figure 1b

We exploit this feature to implement a bounded software model checker that unwinds the program *incrementally* and terminates as soon as an assertion violation can be detected – even if the specified unwinding bound is not reached. Our method eliminates the necessity for restarts and avoids the construction of unnecessarily large SMT instances. Moreover, it relieves the user of its responsibility to provide an adequate bound. While incremental solving necessarily results in a computational overhead, the performance penalty is quickly offset by the avoidance of restarts in situations where the exact bound is not known. In addition (and unlike non-incremental tools), our technique is able to provide partial results for bug-free programs even if the specified bound is prohibitively large.

We present our incremental BMC method in Section 2 and provide an experimental evaluation in Section 3. Our implementation is available under the terms of the GNU public license version 3 (GPL3) on github (`https://github.com/hguenther/nbis`).

2. INCREMENTAL BMC

2.1 Programs

A program is a directed graph $\langle \mathsf{Locs}, \mathsf{Stmts} \rangle$ with nodes Locs (representing the program locations including the initial location $l_0 \in \mathsf{Locs}$) and edges Stmts annotated with guarded assignments $\langle [\gamma], \mathtt{x} := e \rangle$, where the guard γ is a predicate over the program variables, and e is an expression assigned to variable $\mathtt{x}$. The guard may be omitted if it is true and the assignment may be omitted if the edge is a conditional jump. The semantics of guarded assignments is determined by the predicate transformer

$$sp(\langle [\gamma], \mathtt{x} := e \rangle, \varphi) \stackrel{\text{def}}{=} \exists \mathtt{x}_i \,.\, (\varphi \wedge \gamma)[\mathtt{x}/\mathtt{x}_i] \wedge (\mathtt{x} = e[\mathtt{x}/\mathtt{x}_i]) \,,$$

where i is a fresh index and $\varphi[\mathtt{x}/\mathtt{x}_i]$ denotes the formula φ with all free occurrences of $\mathtt{x}$ replaced by $\mathtt{x}_i$.

An *unwinding* of a program $\langle \mathsf{Locs}, \mathsf{Stmts} \rangle$ is a connected DAG $\langle V, E \rangle$ with nodes V and edges E such that there exists a mapping $\ell : V \to \mathsf{Locs}$ and for every $\langle v_1, v_2 \rangle \in E$ we have $\langle \ell(v_1), \ell(v_2) \rangle \in \mathsf{Stmts}$, and each edge $\langle v_1, v_2 \rangle \in E$ is accordingly annotated with a guarded assignment $\mathsf{stmt}(v_1, v_2)$. Moreover, there is a unique root node $v_0 \in V$ with $\ell(v_0) = l_0$. Figures 1b and 1c show unwindings for the program in Figure 1a. Given an unwinding $\langle V, E \rangle$, the formulas ψ_v representing reachable states for each node $v \in V$ are defined inductively:

$$\psi_v \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \text{if } v = v_0 \\ \bigvee_{\langle u,v \rangle \in E} sp(\mathsf{stmt}(u,v), \psi_u) & \text{otherwise} \end{cases} \qquad (1)$$

This symbolic representation can be derived from the SSA form of an unwinding in a straight forward manner. An assertion $\mathtt{assert}(\alpha)$ at node $v \in V$ can be violated if $\psi_v \wedge \neg\alpha$ is satisfiable, which can be easily checked using an SMT solver.

2.2 Merge Nodes

The iterative expansion of a given unwinding $\langle V, E \rangle$ results in new loop exit edges incident to the node succeeding the loop (node u in Figure 1, for instance). These nodes, which we call *merge nodes*, are chosen in a manner such that the resulting expanded unwinding remains cycle-free. Expanding the unwinding increases the in-degree of merge nodes $v \in V$ and necessitates a modification of the corresponding predicate ψ_v defined above (1).

While contemporary SMT solvers allow for adding additional conjuncts to the formulas ψ_v (1), an expansion of the disjunctions for merge nodes v is not possible, thus requiring a reconstruction of ψ_v and a restart of the solver.

2.3 Incremental Representation

To avoid the problem described above, we use a symbolic representation of unwindings $\langle V, E\rangle$ that can be extended on demand. An SSA representation of $\langle V, E\rangle$ guarantees that exactly one version of each program variable is associated with each node v (e.g., $\mathtt{c}_2$ and $\mathtt{y}_2$ at v_1 and $\mathtt{c}_3$ at u in Figure 1b). We use $\mathtt{x}_v$ to denote the SSA version of $\mathtt{x}$ in scope at node v.

For each node v, we introduce a propositional activation variable a_v which indicates whether the unwinding contains a feasible execution path reaching v. Given an unwinding $\langle V, E\rangle$ in SSA form, a_v is constructed as follows:

$$a_v \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \text{if } v = v_0 \\ \begin{array}{l} p_v \vee \bigvee_{\langle u,v\rangle \in E} (a_u \wedge \gamma) \\ (\text{where } \mathsf{stmt}(u,v) = \langle [\gamma], \_\rangle) \end{array} & \text{otherwise} \end{cases}$$

For node u in Figure 1b, for instance, we obtain

$$a_u = p_u \vee (a_{v_0} \wedge (\mathtt{y}_1 = 0)) \vee (a_{v_1} \wedge (\mathtt{y}_2 = 0))\,. \qquad (2)$$

The disjunct p_v is an optional *proxy* variable which is only introduced at merge nodes. Proxy variables enable us to retroactively introduce additional incoming edges in the encoding. This technique is similar to an encoding used for LTL [17]. The expansion of the unwinding in Figure 1b to the unwinding in Figure 1c results in the constraint $p_u = (a_{v_2} \wedge (\mathtt{y}_3 = 0) \vee p_w)$, where p_w is a fresh proxy variable. Whenever we call the SMT solver, "dangling" proxy variables are constrained by adding a *retractable* formula $\neg p_w$.

Unlike in Formula 1, assignments are modeled as separate constraints:

$$\bigwedge \{\mathtt{x}_v = e \,|\, \langle u,v\rangle \in E \wedge \mathsf{stmt}(u,v) = \langle \_, \mathtt{x}_v := e\rangle\}$$

Nodes for which the variable versions of the incoming edges disagree are annotated with ϕ functions in SSA (e.g., $\mathtt{c}_3 = \phi(\mathtt{c}_1, \mathtt{c}_2)$ for node u in Figure 1b). A ϕ function for $\mathtt{x}$ at node v is encoded as

$$\bigwedge_{\langle u,v\rangle \in E} ((a_u \wedge \gamma) \Rightarrow (\mathtt{x}_v = \mathtt{x}_u)) \text{ (where } \mathsf{stmt}(u,v) = \langle [\gamma], \_\rangle) \qquad (3)$$

(assuming that $\mathsf{stmt}(u,v)$ does not update $\mathtt{x}_u$). Formula 3 can be augmented upon expansion of the unwinding. The encoding of $\mathtt{c}_3 = \phi(\mathtt{c}_1, \mathtt{c}_2)$ at node u in Figure 1b is $(a_{v_0} \wedge (\mathtt{y}_1 = 0) \Rightarrow (\mathtt{c}_3 = \mathtt{c}_1)) \wedge (a_{v_1} \wedge (\mathtt{y}_2 = 0) \Rightarrow (\mathtt{c}_3 = \mathtt{c}_2))$, to which $a_{v_2} \wedge (\mathtt{y}_3 = 0) \Rightarrow (\mathtt{c}_3 = \mathtt{c}_4)$ is added upon further unwinding (Figure 1c).

Assertions are represented using propositional variables b_v which are constrained with the negated assertion condition and the activation variable of the respective node. The assertion in Figure 1 yields the constraint

$$b_u = a_u \wedge ((\mathtt{x}\&42 = 42) \wedge (\mathtt{c}_3 \leq 7))\,.$$

To check for assertion violations, we assert a disjunction over all assertion variables.

2.4 Pointers and Memory

Dynamic memory accesses are implemented by maintaining a set of *memory states* Mem and a set of *pointers* Ptrs. Each memory state represents the state of the memory at a certain point of program execution and contains a set of all the allocated *memory objects*. Each memory object is represented by a bitvector SMT-variable and has a unique identifier. For a memory-state $m \in \mathsf{Mem}$, we write $m(i)$ to refer to the memory object identified by i. Every pointer $p \in \mathsf{Ptrs}$ has two attributes:

- A set of object identifiers $\mathsf{points\text{-}to}(p)$ which keeps track of all the objects the pointer p can potentially point to. This is required to limit the number of case splits over the objects the pointer can actually point to, which reduces the strain on the SMT solver.

- The SMT representation of the actual object identifier the pointer points to, refered to as $\mathsf{repr}(p)$. This can be any SMT expression such that it evaluates to one of the object identifiers in its *points-to* set, i.e.

$$\bigvee_{i \in \mathsf{points\text{-}to}(p)} \mathsf{repr}(p) = i$$

  always holds.

In the following, we introduce a set of *memory instructions* which we use to encode constraints over Mem, Ptrs, and the program variables. As the program is unwound incrementally, the program statements are converted into memory instructions, which are then applied to successively add the corresponding constraints to the encoding.

- **connect** $c\, m_1\, m_2$ enforces the conditional equivalence of the memory states m_1 and m_2. This is achieved by generating constraints such that

$$c \Rightarrow (\forall i \,.\, m_1(i) = m_2(i))$$

  holds. Note that the quantifier can be expanded, since the number of objects i is finite.

- **connect_ptr** $c\, p_1\, p_2$ connects the pointers p_1 and p_2 if the condition c holds. The memory model must generate constraints to make $c \Rightarrow \mathsf{repr}(p_1) = \mathsf{repr}(p_2)$ and $\forall i \in \mathsf{points\text{-}to}(p_1).i \in \mathsf{points\text{-}to}(p_2)$ true.

- **alloc** $m_1\, p\, s\, m_2$ allocates a new object of size s and creates a new state m_2, which contains the new object and all previous objects of m_1. The pointer p is initialized to point to the new object. Accordingly, for the fresh object identifier i and the SMT bitvector variable v representing the new object, the instruction yields the following:

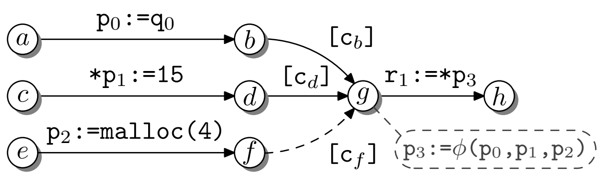
$$\begin{gathered} \forall i' \neq i \,.\, m_2(i') = m_1(i'), \\ m_2(i) = v, \mathsf{repr}(p) = i, \text{ and} \\ \mathsf{points\text{-}to}(p) = \{i\}\,. \end{gathered}$$

- **null** p constrains the pointer p to point to the *null* object. The representation of the *null* object is 0 and the points-to set is empty:

$$(\mathsf{repr}(p) = 0) \wedge (\mathsf{points\text{-}to}(p) = \emptyset)$$

- **load** $m\, p\, r$ assigns the content of pointer p in state m into the SMT variable r. The encoding guarantees that if the representation of the pointer p matches an object identifier i then the result r will be the memory object $m(i)$:

$$\bigwedge_{i \in \mathsf{points\text{-}to}(p)} (\mathsf{repr}(p) = i) \Rightarrow (r = m(i))\,.$$



(a) CFG

$$\left.\begin{array}{l}\textbf{connect\_ptr}\, c_b\, q_0\, p_3\\ \textbf{connect}\, c_b\, m_b\, m_g\\ \textbf{store}\, m_c\, p_1\, 15\, m_d\\ \textbf{connect\_ptr}\, c_d\, p_1\, p_3\\ \textbf{connect}\, c_d\, m_d\, m_g\\ \textbf{load}\, m_g\, p_3\, r_1\end{array}\right\}\text{step 1}$$

$$\left.\begin{array}{l}\textbf{alloc}\, m_e\, p_2\, 4\, m_f\\ \textbf{connect\_ptr}\, c_f\, p_2\, p_3\\ \textbf{connect}\, c_f\, m_f\, m_g\end{array}\right\}\text{step 2}$$

(b) Memory instructions

Figure 3: Incremental memory instructions

- **store** $m_1\, v\, p\, m_2$ stores the value v into the object that the pointer p is pointing to at state m_1, thus creating a new memory state. The constraint

  $$m_2(i) = \begin{cases} v & \text{if } i \in \mathsf{points\text{-}to}(p) \wedge \mathsf{repr}(p) = i \\ m_1(i) & \text{otherwise} \end{cases}$$

  is maintained for all objects i.

- **compare** $p_1\, p_2\, r$ compares the pointer p_1 and p_2 for equality and stores the result in the SMT variable r, such that $r = (\mathsf{repr}(p_1) = \mathsf{repr}(p_2))$ holds.

Since the points-to sets of pointers can change due to new *connect*-instructions, load and store instructions from or to these changed pointers have to be augmented to take the newly reachable objects into account. Consider the example given in Figure 3. Every node v in the CFG has a memory state m_v associated with it. Since the assignment $\mathsf{p}_0\mathsf{:=q}_0$ does not manipulate the dynamically allocated memory, we have $m_a = m_b$. At the $\langle a, b\rangle$ edge, the address of q_0 gets assigned to p_0, so we use q_0 instead of introducing an alias pointer.

To realize the edge $\langle b, g\rangle$, we must first connect the pointer q_0 to the pointer p_3 (associated with node g). This means that we have to initialize the *points-to* set for p_3 with the one of q_0. Suppose that q_0 can point to two object identifiers i_0 and i_1, which are represented in m_b by the objects $m_b(i_0)$ and $m_b(i_1)$. We obtain $\mathsf{points\text{-}to}(p_3) = \{i_0, i_1\}$. We also get the following constraint for the SMT instance: $c_b \Rightarrow \mathsf{repr}(p_3) = \mathsf{repr}(q_0)$. Then we have to connect the memory state m_b to m_g, which yields $c_b \Rightarrow m_g(i_0) = m_b(i_0)$ and $c_b \Rightarrow m_g(i_1) = m_b(i_1)$.

Suppose that p_1 can only point to the object identified by i_0. Storing 15 to this pointer creates a memory state m_d in which $m_d(i_0) = 15$. Connecting p_1 to p_3 does not change the *points-to* set of p_3 since i_0 is already contained in it. However, we get the new SMT constraint $c_d \Rightarrow \mathsf{repr}(p_3) = \mathsf{repr}(p_1)$. By connecting m_d to m_g, we only get one constraint, namely $c_d \Rightarrow m_g(i_0) = m_d(i_0)$, since m_d does not contain i_1.

Loading from pointer p_3 at edge $\langle g, h\rangle$ entails adding a constraint for each object identifier in $\mathsf{points\text{-}to}(p_3)$: $\mathsf{repr}(p_3) = i_0 \Rightarrow r = m_g(i_0)$ and $\mathsf{repr}(p_3) = i_1 \Rightarrow r = m_g(i_1)$.

Now the third incoming edge is added in step 2; suppose that m_e is empty, so the allocation creates a fresh variable v of 4 bytes and a fresh identifier i_2 such that $m_f(i_2) = v$. The pointer p_2 is created with the singleton *points-to* set $\mathsf{points\text{-}to}(p_2) = \{i_2\}$ and the representation $\mathsf{repr}(p_2) = i_2$. Connecting the pointers p_2 and p_3 now adds the new object identifier i_2 into the *points-to* set of p_3, so we have to augment the SMT formulas generated by the load instruction from p_3 by the following formula: $\mathsf{repr}(p_3) = i_2 \Rightarrow r = m_g(i_2)$. The constraints $c_f \Rightarrow \mathsf{repr}(p_3) = \mathsf{repr}(p_2)$ and $c_f \Rightarrow m_g(i_2) = m_f(i_2)$ are added as before. After adding these constraints r now represents every possible loading result of the three incoming edges.

2.5 Catching Memory Bugs

To detect invalid memory accesses (either loads from or stores to null-pointers), we must generate an assertion $\mathsf{repr}(p) \neq 0$ for every memory instruction **load** $m\, p\, r$ or **store** $m_1\, v\, p\, m_2$. However, oftentimes we can actually statically infer that p can never be a null pointer. We can accomplish this by introducing a special object identifier i_{null} which identifies no allocated object but instead its presence in a *points-to* set signifies that the pointer can potentially be null. With this in place, we only have to generate the assertions for pointer with $i_{null} \in \mathsf{points\text{-}to}(p)$.

2.6 Extending the Memory Model

The memory model described above is very limited and only able to handle very simple programs without arrays, pointer indirections, casts or structs. We informally describe the various extensions implemented in NBIS to handle more complex programs here:

- *Pointer stores and loads.* To enable the memory model to store and load pointers to/from other pointers we need to extend each memory object with a *points-to* set. Whenever a pointer is loaded from a memory object, it inherits its *points-to* information from the memory object. Similarily, storing a pointer transfers its *points-to* information to the memory-object.

- *Structures.* Instead of representing each memory model with one single SMT variable, we can allow a memory object to be composed of multiple SMT variables, where each variable represents a field in structure data type.

- *Arrays.* While arrays of a constant size can be handled by creating an SMT variable for each array element, arrays with a variable size require more thought. We can represent arrays of dynamic size using the SMT theory of arrays with McCarthy's `select` and `update` functions to manipulate arrays. Each array is represented by an SMT array variable representing the content of the array and a bitvector variable storing the size of the array for error checking (if the array index is larger than the size variable, we detect a memory access violation).

Since LLVM handles all arrays as heap objects, we have to augment the symbolic representation of pointers. Instead of having the pointer representation only identify the object the pointer is pointing to, we split the pointer representation into two parts: The first part of the pointer represents the object identifier, as before, while the second half of the pointer representation can be used to represent a potential offset into the object. To avoid having to check for all possible offsets into a given object, we can also augment the *points-to* set of pointers with a set of offsets that the pointer can potentially represent.

- *Global variables.* Since global variables are implemented in LLVM as pointers to pre-allocated objects, a global variable v can be represented by an object identifier i_v which is present in every memory state and a pointer p_v which only points to the object identified by i_v. We generate a memory instruction $\mathbf{alloc}\, m_0\, p_v\, s\, m_{start}$ at the beginning of the unrollment where s is the size of the global variable and m_{start} is the initial memory state for the program.

- *Pointer casts.* Since the C-language allows *almost* every possible conversion between pointers, care has to be taken to incorporate pointer casts into the memory model. For example, if the program casts a pointer to a 64-bit integer into a byte-array and accesses the pointer using a dynamic offset, the loading instruction has to generate a case split over all the byte components of the integer.

2.7 Optimizations

Since increasing the bound may add new incoming edges to merge nodes, it is not possible to safely infer information about the values of variables from a given unwinding. Accordingly, optimizations such as constant-propagation, elimination of overflow-checks, etc. can only be applied by performing an up-front static analysis of the program. We perform an approximate static analysis to infer the following information: (a) lower and upper bounds of variables to remove redundant array-bounds checks, (b) access and alignment information for data structures to simplify load and store instructions, (c) alias information to remove redundant checks for null-pointer accesses.

3. EVALUATION

To evaluate our approach, we implemented it in a tool called Nbis, written in Haskell. Nbis uses the intermediate representation of the LLVM compiler framework [19], which simplifies the handling of the complex semantics of the C programming language. Our implementation supports a range of SMT solvers such as Z3 [12], MathSAT [8], STP [16], CVC4 [2], Yices [14], and others supporting the SMT-LIB standard [3]. The implementation is available under the terms of the GNU public license version 3 (GPL3) on github (`https://github.com/hguenther/nbis`).

To demonstrate the feasibility of incremental verification, we evaluated Nbis on the programs in the bitvector category of the SV-Comp 2013.[1] First, we compared Nbis in non-incremental mode to the state-of-the-art tools CBMC [9], ESBMC [10], and LLBMC [21]. Since CBMC relies on bit-blasting and a *SAT*-solver, we compared it to Nbis running with the STP [16] backend. We also compared Nbis in this configuration with LLBMC, since it also uses STP as its backend. ESBMC, on the other hand, uses Z3 [12] as a solver, so we used the same solver as the backend in our comparison. Figure 4 shows the running times of these tools plotted with a logarithmic time-scale. The performance of Nbis running with STP is comparable (and often even better) than CBMC and only slightly worse than LLBMC. Comparing Nbis with ESBMC, we can see that Nbis fares better on every benchmark.

To measure the performance overhead of incremental BMC, we ran Nbis on every benchmark with different SMT-backends and compared the performance to the running time in non-incremental mode. A fair comparison between incremental and non-incremental BMC is difficult, because the run-time is influenced by the following parameters:

1. *Unwinding depth.* In the presence of a bug, the non-incremental approach is at a disadvantage if the unwinding depth significantly exceeds the depth at which the bug manifests itself.
2. *Check interval.* By default, Nbis checks for bugs after each unwinding, resulting in a significant overhead if each unwinding step only adds a small number of constraints to the instance. By increasing the number of unwindings after which a check is performed to the unwinding depth, we can enforce that only one SAT query is made, which is equal to the non-incremental algorithm.

Since it is always possible to tweak these parameters in favor of either incremental or non-incremental BMC, we measure the *worst* case for incremental verification:

- The bound for the non-incremental is set to the minimal depth where the bug appears. If no bug is present and a completeness threshold can be computed, it is used as the bound. Otherwise a bound of 10 is selected.
- The incremental algorithm checks for bugs after each unrolling step.

Table 1 shows the overhead of running the incremental algorithm on the problem instances, where an overhead of n means that the incremental version took n times as long to complete. Missing entries indicate a time-out, which was set to 30 seconds. The smallest overhead is highlighted.

We make the following observations:

1. The performance of incremental verification is contingent on the solver: There are many examples—such as "gcd 3"—where some solvers perform significantly better than the rest.

2. Many examples from the bitvector benchmark suite show a less-than twofold increase in execution time, even under the worst possible circumstances. This is very encouraging, as it suggests that the approach is indeed viable for a wide range of examples.

3. Large overheads (such as the 12-fold increase of running time in the "modulus safe" benchmark) are owed to the fact that incremental BMC prevents constant propagation in the unwinding. This problem can be mitigated by performing an up-front static analysis to detect constants. We will add this feature in a future version of Nbis.

[1] We used the following versions of solvers: Z3 4.3.1, STP 1d89673988c7d86fc3bca1d0ab9a7497366bab04, MathSAT 5.2.10, CVC4 1.4-prerelease and Yices 2.1.0

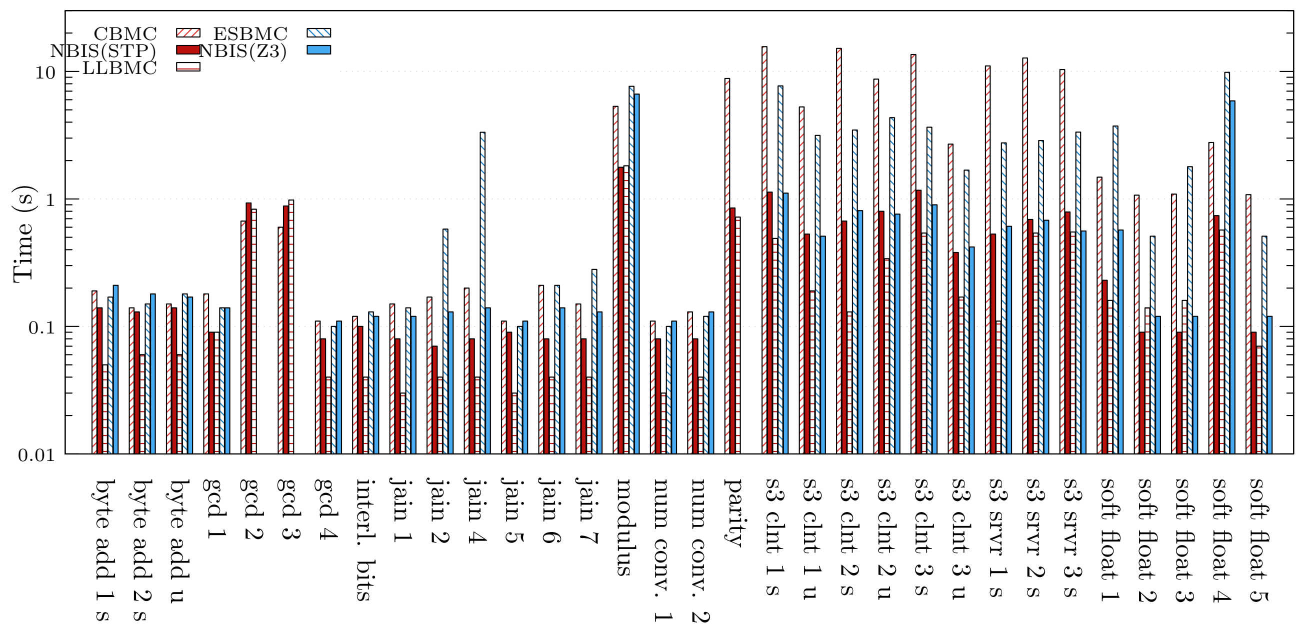


Figure 4: Non-incremental verification times

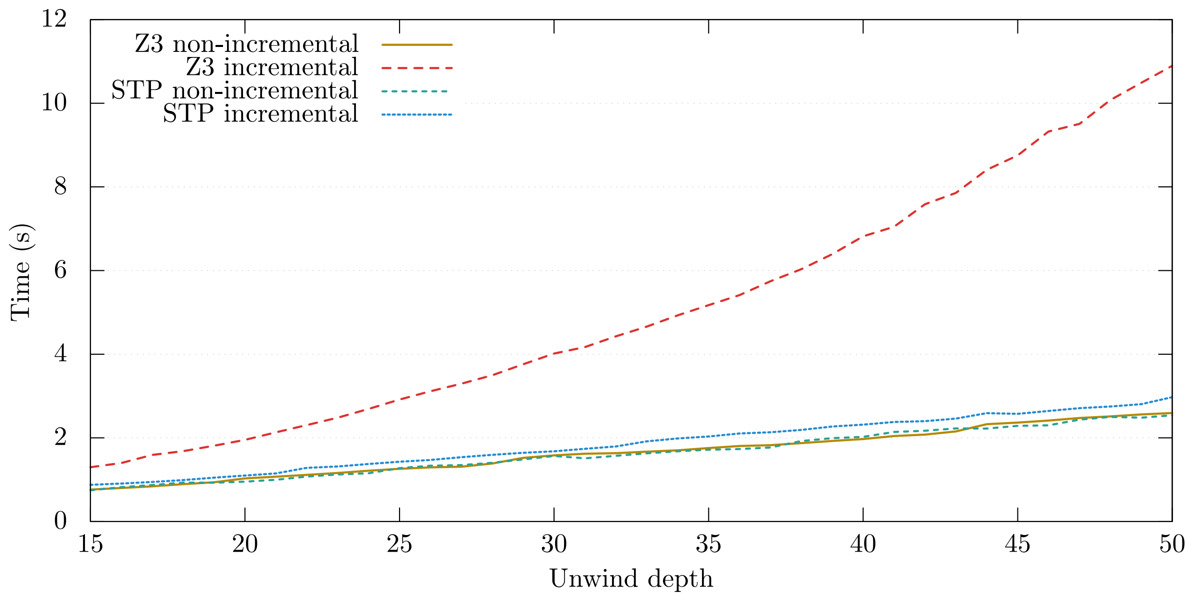


Figure 5: Runtime comparison on the "s3 clnt 2 safe" benchmark

Figure 5 illustrates the runtime variation resulting from the different performance characteristics of the SMT solvers: We ran Z3 and STP on the "s3 clnt 2 safe"-benchmark and compared the running times for each unwinding depth. While STP has a linear increase in running time both in incremental and non-incremental mode, Z3 shows a much steeper curve in incremental mode. As of yet, the reasons for these differences between solvers are unknown to the authors.

Figure 5 also illustrates that the additional cost of incremental verification amortizes quickly once the non-incremental solver is restarted for the first time: For the given example, the overhead of the incremental algorithm is never larger than the cost of restarting the non-incremental algorithm.

4. RELATED WORK

A number of verification tools, such as CBMC [9], ESBMC [10], and LLBMC [21], F-Soft [18], SMT-BMC [1] are based on non-incremental BMC. CBMC performs bit-blasting and uses the SAT solver MiniSAT [15] to solve the resulting propositional problem, while SMT-BMC and ESBMC deploy an SMT solver. ESBMC and CBMC use the same front-end for parsing C-files. F-Soft stands out as it performs several static analyses on the program in order to simplify the resulting unwinding instance. LLBMC bears the closest similarity with Nbis, since it also uses the LLVM internal representation. None of these tools allow the bound to be increased incrementally.

Table 1: Incremental verification time overhead

Benchmark	Z3	STP	MathSAT	Boolector	CVC	Yices
byte add 1	**6.8**	7.6	54.9	11.5	11.5	10.7
byte add 2	**6.5**	8.2	39.1	15.0	12.6	14.8
byte add u	**5.1**	5.8	9.2	47.0	–	6.0
gcd 1	1.1	**1.0**	3.0	**1.0**	**1.0**	**1.0**
gcd 2	–	5.7	3.0	–	**1.3**	–
gcd 3	–	7.8	2.7	–	**1.3**	–
gcd 4	1.0	1.0	1.0	1.0	1.0	1.0
interl. bits	1.3	1.1	1.1	1.2	1.1	**1.0**
jain 1	1.1	1.3	**1.0**	1.1	**1.0**	**1.0**
jain 2	**1.0**	**1.0**	**1.0**	1.3	**1.0**	1.1
jain 4	1.3	**1.0**	1.1	1.4	**1.0**	**1.0**
jain 5	1.0	1.0	1.0	1.0	1.0	1.0
jain 6	1.3	1.1	**1.0**	1.3	**1.0**	**1.0**
jain 7	1.2	**1.0**	**1.0**	1.1	1.1	**1.0**
modulus	–	**11.7**	–	–	–	–
num conv. 1	1.1	1.1	**1.0**	1.1	1.1	**1.0**
num conv. 2	1.2	**1.0**	1.1	1.1	1.1	**1.0**
parity	–	14.9	–	**14.4**	–	–
s3 clnt 1 s	5.2	6.7	10.5	2.6	–	**2.5**
s3 clnt 1 u	1.7	2.3	2.3	**1.4**	–	1.8
s3 clnt 2 s	1.8	**1.1**	2.0	2.0	8.5	1.2
s3 clnt 2 u	2.7	4.7	6.9	**1.9**	–	2.2
s3 clnt 3 s	5.7	7.68	13.5	2.2	–	**2.0**
s3 clnt 3 u	**1.3**	2.3	1.6	1.4	–	1.6
s3 srvr 1	1.7	**1.3**	1.6	1.9	6.5	1.5
s3 srvr 2	3.1	4.3	**1.6**	1.9	–	1.9
s3 srvr 3	3.5	1.8	4.6	**1.1**	–	1.3
soft float 1	–	–	–	–	–	–
soft float 2	1.0	1.0	1.0	1.0	1.0	1.0
soft float 3	**1.0**	**1.0**	**1.0**	**1.0**	1.23	**1.0**
soft float 4	–	–	–	–	–	–
soft float 5	**1.0**	1.1	1.1	1.1	1.1	1.1

Symbolic execution tools like KLEE [7] or KLOVER [20], on the other hand, incrementally unwind the paths of a program. Since these tools are typically aimed at test case generation, they aim at satisfying coverage criteria. Our approach avoids the explicit enumeration of paths by performing a block-wise unwinding that encodes all program paths in a single SMT instance by aggressively merging similar states. LAV [24], a recent LLVM-based addition to the BMC family, performs a block-wise unwinding of loops, but does not merge the branches of the unwinding. As an additional feature, LAV is able to over-approximate loops (at the cost of potential false alarms). Aggressive merging of (loop-free) paths has also proved beneficial in the context of abstraction and unbounded model checking [4].

Many symbolic execution tools are performing a different kind of incremental verification from our tool: Instead of incrementing the search depth and re-using already learned facts from previous depths, they try to use facts learned from verifying a previous version of the same program with small changes [22][26]. Other symbolic execution techniques focus on maintaining a database of already learned facts to facilitate information re-use between runs [23].

5. CONCLUSION AND FUTURE WORK

We introduced a BMC approach which takes advantage of incremental SMT solvers in order to perform a gradual unwinding of the program. Incremental BMC is favorable if the specified unwinding depth significantly exceeds the depth of the bug and relieves the user of the burden to determine an adequate bound. In addition, an incremental BMC can report partial results for bug-free programs even if the specified unwinding depth is not reached.

As the presented benchmarks show, the performance of incremental bounded model checking is encouraging on many examples. We are confident that the overhead for the remaining examples can be addressed with additional optimizations (such as an up-front static analysis enabling constant propagation) in future versions of Nbis.

In addition, incremental BMC enables additional optimizations typically used in symbolic simulation: the ability to perform a query at any point during the unwinding process enables the verification tool to prune infeasible traces. This optimization will be incorporated into a future version of Nbis.

6. ACKNOWLEDGEMENTS

This work was supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005.

7. REFERENCES

[1] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Software Tools for Technology Transfer (STTT)*, 11(1):69–83, Jan. 2009.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[3] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Workshop on Satisfiability Modulo Theories*, 2010.

[4] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 25–32. IEEE, 2009.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[6] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, 2009.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation (OSDI)*, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[8] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.

[9] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988, pages 168–176. Springer, 2004.

[10] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering (TSE)*, 38(4):957–974, July 2012.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[12] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, LNCS, pages 337–340. Springer, 2008.

[13] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[14] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, August 2006.

[15] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919, pages 502–518. Springer, 2004.

[16] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 519–531. Springer, 2007.

[17] K. Heljanko, T. Junttila, and T. Latvala. Incremental and complete bounded model checking for full pltl. In K. Etessami and S. Rajamani, editors, *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 98–111. Springer, 2005.

[18] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science (TCS)*, 404(3):256–274, Sept. 2008.

[19] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization (CGO)*, 2004.

[20] G. Li, I. Ghosh, and S. Rajan. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 609–615. Springer, 2011.

[21] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In R. Joshi, P. Müller, and A. Podelski, editors, *International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 7152 of *LNCS*, pages 146–161. Springer, 2012.

[22] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 504–515. ACM, 2011.

[23] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Foundations of Software Engineering (FSE)*, pages 58:1–58:11. ACM, 2012.

[24] M. Vujosevic-Janicic and V. Kuncak. Development and evaluation of LAV: An SMT-based error finding platform – system description. In *International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 7152 of *LNCS*, pages 98–113. Springer, 2012.

[25] P. Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5), May 1960.

[26] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 144–154. ACM, 2012.