Over-approximating loops to prove properties using bounded model checking

Priyanka Darke, Bharti Chimdyalwar, R. Venkatesh, Ulka Shrotri, Ravindra Metta

Tata Research Development and Design Center
{priyanka.darke, bharti.c, r.venky, ulka.s, r.metta}@tcs.com

***Abstract*—Bounded Model Checkers (BMCs) are widely used to detect violations of program properties up to a bounded execution length of the program. However when it comes to proving the properties, BMCs are unable to provide a sound result for programs with loops of large or unknown bounds. To address this limitation, we developed a new loop over-approximation technique LA. LA replaces a given loop in a program with an abstract loop having a smaller known bound by combining the techniques of *output abstraction* and a novel *abstract acceleration*, suitably augmented with a new application of induction. The resulting transformed program can then be fed to any bounded model checker to provide a sound proof of the desired properties. We call this approach, of LA followed by BMC, as LABMC.**

We evaluated the effectiveness of LABMC on some of the SV-COMP14 loop benchmarks, each with a property encoded into it. Well known BMCs failed to prove most of these properties due to loops of large, infinite or unknown bounds while LABMC obtained promising results. We also performed experiments on a real world automotive application on which the well known BMCs were able to prove only one of the 186 array accesses to be within array bounds. LABMC was able to successfully prove 131 of those array accesses to be within array bounds.

I. Introduction

Bounded model checkers (BMCs) have been successfully used to detect property violations in safety critical software.In the Software Verification Competition 2014 (SV-COMP14) [3] the best performing tools were based on bounded model checking. The BMCs - CBMC [5], ESBMC [1], and LLBMC [2] were successful in finding violations of invalid properties explicitly encoded in the programs. To find a violation of a property ϕ in a program P, BMC searches for a counterexample to ϕ, in executions of P bounded by some integer k. If there is no counterexample to ϕ within the bound k, then k has to be increased iteratively until either a counterexample is found or sufficiently long execution paths have been explored to conclude that the property is valid. As k increases, the state space to be explored by the BMC explodes exponentially. Therefore, BMCs do not scale to large k.

In practice, programs often contain loops with infinite, non-computable, or large bounds. For proving properties dependent on such loops, k will also be correspondingly large. Therefore, BMCs often do not scale to verify such loop-dependent properties. For instance, as described in the experimental Section III, BMCs were unable to prove loop-dependent properties in most of the safe programs of SV-COMP14 loop benchmarks[1].

Consider the example C code in Fig 1a, an adapted version of a code snippet from an industrial automotive application. The property ($a == b \;\&\&\; c == 2000$) to be proved is asserted at line 12 and it is a valid assertion. CBMC, ESBMC, and LLBMC are unable to prove this property due to the loops on lines 3 and 5. Moreover, other tools implementing various model checking techniques such as predicate abstraction based model checkers SATABS [6], and BLAST [4]; IMPACT based model checker IMPARA [14]; and k-induction based verifier K-Inductor [8] also could not prove the property.

We present a technique LABMC (Loop Abstraction for Bounded Model Checking) to prove properties of such programs. LABMC replaces a given loop in a program with an over-approximated loop having a smaller known bound, then it feeds this resulting program to a BMC. Our two key contributions in over-approximating loops are, a novel abstract acceleration technique which computes the closed form assignment of linear modifications to a variable in a loop, and a novel application of induction when the property to be verified lies inside a loop. We further combine these two ideas with output abstraction presented in [7]. The resulting transformed program, being an over-approximation of the original program with smaller loop bounds, allows BMCs to prove valid properties (please refer to Section II-F for details).

We evaluated the effect of applying LABMC on the SV-COMP14 loop benchmarks using the competition's best performing BMCs for C programs - CBMC, LLBMC, and ESBMC. Loop abstraction (LA) enabled the successful verification of the benchmark programs by all the BMCs. To assess the usefulness of LABMC in practice, we analyzed an industrial automotive application for array index out of bound property. LABMC outperformed other BMCs and also other model checkers (based on different techniques) by proving more than 70% of the array accesses to be safe.

Scope of LABMC: LABMC today does not handle programs in which the contents of array and memory locations referenced by pointers are modified inside a loop.

The key contributions of this paper are:

- A novel technique of abstract acceleration.

[1]In the SV-COMP14 competition, BMCs- ESBMC and LLBMC were run by specifying an unwinding bound for each loop along with the option $no-unwinding-assertions$ for ESBMC and $no-max-loop-iterations-checks$ for LLBMC; the options used for CBMC were not mentioned.

978-3-9815370-4-8/DATE15/©2015 EDAA

- A novel application of induction when the property to be verified lies inside the loop.
- A novel loop over-approximation technique that combines ideas of output abstraction with the above two ideas, to enable a BMC to prove properties of programs that it cannot otherwise prove due to large and unbounded loops.

II. Loop abstraction technique

In this section we present in detail how we combine output abstraction, abstract acceleration, and a novel application of induction to abstract loops. We first introduce the notations we have used and a few definitions.

A. Notations and definitions

We use the following notations and definitions throughout the paper:

- c denotes the loop condition;
- L denotes the loop body;
- I is the set of variables called inputs to the loop that are read in L or c but not modified;
- IO is the set of variables called input-output or I/O variables of the loop that are read as well as modified in L;
- PO is the set of of variables called pure outputs of the loop that are modified but not read within L;
- O is the set of variables called outputs of the loop that are modified in L, $O = IO \cup PO$;
- $|O|$ denotes the size of O.

It is to be noted that only variables in O are modified by the loop and the values of these variables depend only on variables in $I \cup IO$ and not on variables in PO. This fact is exploited in the proposed abstractions.

B. Output abstraction

A simplistic, imprecise way to abstract loops is to replace it with nondeterministic assignments to all output variables. In [7], Darke et. al have presented another technique to improve the precision of pure output variables, as compared to the simplistic approach. They replace the original loop with an abstract loop having a bound of a known value based on the number of variables modified in the loop body. We refer to this abstraction as *output abstraction*. It is presented in Fig. 2 for the input code of Fig. 1e. The abstract loop body assigns nondeterministic values to the I/O variables such that the loop condition holds, followed by the original loop body (lines 2-4 in Fig. 2). The new loop iterates at most as many times as the number of output variables in the loop, shown by the for loop of $|O|$ iterations (line 1). After this loop, c does not hold (ensured by $assume(!c)$ on line 6). This ends the loop abstraction and the remaining input program is kept as it is. It should be noted that for a given assumption $assume(c)$ the model checker considers only those runs for which c holds at that point. If c does not hold at that point for any execution of the program, then the model checker reports it to be unreachable. For example, the program point of $assume(0)$ will be unreachable for all executions of any program.

As can be seen in Fig. 2, the abstracted loop has a known bound. To see that this is an abstraction, consider

```
1  for(i=0;i<|O| && c;i++){
2    nondet(IO);
3    assume(c);
4    L;
5  }
6  assume(!c);
7  assert(P1);
```

Fig. 2. Output abstraction

an arbitrary run of the original loop consisting of n iterations $[i_1, i_2, \ldots, i_n]$. We construct a run of the abstract loop $[a_1, a_2, \ldots, a_x]$, consisting of $x \leq n$ iterations. Let the abstract iteration a_1 map to iteration i_{j_1}, where i_{j_1} is the first iteration to modify one or more output variables O_{j_1}, that are never modified in subsequent iterations of the arbitrary run. Let the non-deterministic values assigned to I/O variables at the start of a_1 be the same as the values of the I/O variables at the start of i_{j_1}. This implies that the values of variables in O_{j_1} will be the same at the end of i_{j_1} and a_1 because they execute the same body. A similar argument can be extended for a_2 and $i_{j_2}, \ldots$ and a_x and i_{j_x}. Since each i_{j_y} modifies at least one distinct output variable $x \leq |O|$.

In the next section we explain how we improve the precision of output abstraction using abstract acceleration.

C. Abstract acceleration of I/O Variables

The technique of output abstraction uses nondeterministic assignments to I/O variables. Hence it is imprecise. In this paper, we improve the precision by replacing the nondeterministic assignments with accelerated assignments to I/O variables. The abstract loop after applying abstract acceleration is presented in Fig. 1i for the input loop given in Fig. 1e.

Now consider an I/O variable, io, to be accelerated to its actual value, io^k, at the end (start) of the k^{th} ($k+1^{th}$) iteration of the original loop. Based on its usage and modification in the loop body we present how LABMC generates io_a^k, an abstraction of io^k, below.

1) Abstract acceleration of non-recurrent I/O variables: We define io to be *non-recurrent* in the loop body if it is modified under some condition only in expressions of the form $io = \gamma$, also called *reset expressions*. Here γ is an expression only referring to constants or input variables. Then the value of io is accelerated using the following:

$$io_a^k = io^0 \;||\; \gamma_1 \;||\; \gamma_2 \;||\; \ldots \;||\; \gamma_r \tag{1}$$

Where, io^0 is the initial value of io before loop execution, r is the number of reset expressions of io, $\gamma_i, 1 \leq i \leq r$ is the right hand side of the corresponding reset expression of io. In an arbitrary execution of the original loop, io will have a value of io^0 or γ_i at the start of the $k+1^{th}$ iteration. Using equation 1, we can construct an abstract acceleration assignment such that the actual value of io at the start of the $k+1^{th}$ iteration can be generated, thus making the value an abstraction.

2) Abstract acceleration of self recurrent variables: We define io to be *self recurrent* in the loop if it is modified under some condition only in statements of the form $io = io + \beta$,

```
1 func(int last) {
2   int a=0,b=0,c=0,st=0;
3   while(1) {
4     st=1;
5     for(c=0;c<2000;c++)
6       if (c==last ) st = 0;
7     if(st==0 && c==last+1){
8       a+=3; b+=3;}
9     else {  a+=2; b+=2; }
10    if(c==last && st==0)
11      a = a+1;
12    assert(a==b && c==2000);
13  }
14 }
```

(a) Motivating example

```
1 //L contains assert(P2)
2
3   //base case
4   if(c)
5     L; //with assert(P2);
6   //induction hypothesis
7   //k iterations
8   for(i=0;i<|O| && c;i++){
9     accelerate(IO);
10    assume(c);
11    L; //with assume(P2);
12  }
13  //induction step
14  //(k+1)th iteration
15  if(c)
16    L; //with assert(P2);
17  assume(!c);
```

(b) Abstraction with property inside loop

```
1 inductionAbs(){
2   // Base case
3   output("if(c)")
4     output(L); //has assert
5   // kth iteration
6   simpleAbs(L,INDUCTION);
7   // (k+1)th iteration
8   output("if(c)")
9     output(L); //has assert
10 }
```

(c) Induction Abstraction

```
1 func(int last){
2   a=0, b=0, c=0, st=0;
3   a0= a; b0= b; c0= c;
4   /* Base case */
5   if(1){
6     <outer_loop_body>;
7     assert(a==b && c==2000);
8   }
9   /* k th Loop iteration */
10  for (t1=0; t1<4; t1++) {
11    k_1^1 = *; k_1^1 = *; k_2^1 = *;
12    k_3^1 = *;
13    a=a0 + 3*k_1^1 + 2*k_2^1 + k_3^1;
14    b= b0 + 3*k_1^1 + 2*k_2^1;
15    assume(0<k && 0<=k_1^1 &&
16      0<=k_2^1 && 0<=k_3^1 );
17    assume(k_1^1<=k && k_2^1<=k
18      && k_3^1<=k);
19    assume(k == k_1^1 + k_2^1);
20    <outer_loop_body>;
21    assume(a==b && c==2000);
22  }
23  /*(k+1)th Loop iteration*/
24  if(1){
25    <outer_loop_body>;
26    assert(a==b && c==2000) ;
27  }
28  assume(!1);
29 }
```

(d) Abstraction of Motivating example

```
1   while(c) {
2     L;
3   }
4   assert(P1);
```

(e) Property outside loop

```
1 //L contains
2 //assert(P2)
3
4   while(c) {
5     L;
6   }
```

(f) Property inside loop

```
1 //L=Loop body,
2 //c=Loop condition,
3
4 LoopAbstraction() {
5   if(assertion_not_in_L)
6     simpleAbs(L, SIMPLE);
7   else // assertion in L
8     inductionAbs(L);
9   output(assume !(c));
10 }
```

(g) Loop Abstraction Algorithm

```
1 c0 = c;
2 for(t=0; t<2&&c<2000 ;t++){
3   k^0=*
4   assume(0<k^0);
5   c = c0+k^0;
6   assume(c<2000);
7   if (c==last) st=0;
8   c++;
9 }
10 assume(!(c<2000));
```

(h) inner_absloop

```
1   for(i=0;i<|O| && c;i++){
2     accelerate(IO);
3     assume(c);
4     L;
5   }
6   assume(!c);
7   assert(P1);
```

(i) Abstraction with property outside loop

```
1 simpleAbs(flag){
2   genAbstractLoopHeader();
3   accelerateIOs();
4   genAssumes();
5   if(flag == INDUCTION)
6     output(L replacing
7     assertion with assume);
8   else
9     output(L);
10 }
```

(j) Simple Abstraction

```
1 st=1;
2 <inner_absloop>
3 if(st==0 && c==last+1){
4   a+=3; b+=3;
5 }
6 else { a+=2; b+=2;
7 }
8 if( c==last && st==0 )
9   a = a+1;
```

(k) outer_loop_body

Fig. 1. Loop abstraction using LABMC

also called *self recurrence* expressions. Here β is a constant or an input. Then in the abstract loop, io is accelerated using the following:

$$io_a^k = io^0 + \sum_{i=1}^{e} k_i \cdot \beta_i \tag{2}$$

Where, e is the number of expressions that modify io and $k_1, k_2, \ldots, k_e$ are an abstraction of the number of times the respective expression was executed in the original loop. Therefore,

$$k_i \leq k \tag{3}$$

LABMC assigns nondeterministic values to $k_1, k_2, \ldots, k_e$ under the constraint of Relation 3.

In the presence of nested loops, LABMC abstracts the inner loop first and unwinds the abstract inner loop. Then it abstracts the outer loop. This way Relation 3 can be applied in the presence of nested loops also.

For an unconditional expression $io = io + \beta_i$ in the loop body

$$k_i = k \tag{4}$$

Claim 1. Relation 2 generates a sound abstraction of io at the start of the $k + 1^{th}$ iteration.

$Proof$. In an arbitrary run of the original loop, let $n_1, n_2, \ldots, n_e$ be the number of times each self recurrence expression of io, $io = io + \beta_i$ was executed. Then at the start of the $k + 1^{th}$ iteration of the original loop, the actual value of io will be given by

$$io^k = io^0 + \sum_{i=1}^{e} n_i \cdot \beta_i$$

Also, $n_i \leq k$. Since in Relation 2 LABMC assigns nondeterministic values to $k_1, k_2, \ldots, k_e$ under the constraint that $k_i \leq k$, using Relation 2 we can construct an assignment to io_a^k such that the actual value io^k can be generated. Therefore Claim 1 holds.

3) Abstract acceleration of self recurrent variables with reset: In this case, io is modified only using either self-recurrence expressions or expressions of the form $io = \gamma$ under some condition. Then in the abstract loop, io^k is abstracted using the following:

$$io_a^k = io' + \sum_{i=1}^{e} k_i \cdot \beta_i \tag{5}$$

Here $io' = io^0 \mid\mid io' = \gamma_j, 1 \leq j \leq r$, γ_j is an expression referring to only constants or input variables, r is the number of reset expressions of io, e is the number of self-recurrence expressions of io, k_i is the number of times the expression containing β_i was executed in the original loop, after the loop

execution started or since the last reset expression of io was executed, whichever was last. Therefore,

$$k_i \leq k \tag{6}$$

In the abstract loop, LABMC selects the values of k_i nondeterministically. Therefore the value of io_a^k in this case also is an abstraction of io^k. The proof can be derived from the proof of Claim 1.

4) Other I/O Variables: For all other I/O variables, LABMC assigns a nondeterministic value to io^k.

Although we have used the technique explained above for accelerating the values of outputs, it is possible to plug-in other acceleration techniques like the ones proposed by Halbwachs and Gonnord in [10] and by Kroening et. al. in [9].

By Claim 1, the abstract acceleration of I/O variables presented above produces a sound abstraction of their values at the start of an arbitrary $k+1^{th}$ iteration of the original loop. Therefore in output abstraction, replacing the nondeterministic assignments to I/O variables with abstract acceleration generates an abstraction of the input loop. Also, this same technique can be used to abstract any number of iterations and is used in the next section.

D. Loop abstraction using induction

When the assertion lies inside the loop body, LABMC further improves the precision of abstraction explained earlier by applying *mathematical induction*. Fig. 1f contains the input code when the property to be verified lies inside the loop body $P2$. Its abstraction using induction is shown in Fig.1b. As can be seen, LABMC first generates a base case which asserts $P2$ (lines 4-5 of Fig. 1b). Following this it generates the induction hypothesis which assumes $P2$ to hold for an arbitrary k iterations, $k \geq 0$ (for loop on line 8). This for loop is an abstraction of k iterations of the input loop. It is generated similar to when the property lies outside L. But it contains an additional assumption (line 11 in Fig. 1b) meaning that till the end of the k^{th} iteration only those runs will be considered by the model checker in which $P2$ was true. After this, a single loop iteration is output as the induction step where $P2$ is asserted (line 16).

Since there is no abstraction in the base case and the induction hypothesis represents a sound abstraction of k iterations of the loop, the loop abstraction using induction is a sound abstraction of the original loop.

E. Algorithm

Given an input program only loops are transformed using our technique. In case of nested loops, the innermost loop is abstracted first and then its immediate enclosing loop is abstracted and so on till the outermost loop. The abstraction algorithm for a loop is presented in Fig. 1g and is explained below using the example C code in Fig. 1a. In the algorithm, L is the loop body of the loop to be abstracted and c is the loop condition.

1) LoopAbstraction(): In this function *simpleAbs* is called when the property or assertion lies outside L and $inductionAbs$ is called when it is within L.

2) simpleAbs(flag): This function is shown in Fig. 1j. If $simpleAbs$ is called with $flag$ value as $INDUCTION$, the abstract loop body is output with the assert replaced by assume as explained in Section II-D. Otherwise if the $flag$ value is $SIMPLE$ then the loop body is output as is.The functions called from this function are explained below.

3) genAbstractLoopHeader(): A for loop header is output for the abstract loop using the computed loop bound. Accordingly, for the sample C code in Fig. 1a, for the abstract inner and outer loops the bounds are $|O| = 2$ and $|O| = 4$, respectively.

4) accelerateIOs(): The abstract assignments for all I/O variables are generated as explained in Section II-C. In Fig. 1a the inner loop has one self-recurrent variable c and is accelerated using the equation $c = c + 1 * k^0$ and k^0 is assigned a nondeterministic value representing the number of iterations of the original inner loop. The outer loop has two self-recurrent variables a and b, accelerated using assignments $a = a0 + 3 * k_1^1 + 2 * k_2^1 + k_3^1$ and $b = b0 + 3 * k_1^1 + 2 * k_2^1$ respectively. k_1^1, k_2^1, k_3^1 are assigned nondeterministic values and represent the number of times the bodies of if at line 7, $else$ at line 9 and if at line 10 in Fig. 1a were executed in the original outer loop execution. $a0$ and $b0$ are the initial values of a and b respectively.

5) genAssumes(): This function generates assume statements implied by Relations 3 and 6 to constrain the values of I/O variables after acceleration and to improve precision.

6) inductionAbs(): This function is called when the assertion lies inside L and is shown in Fig. 1c. It first outputs the base case, then the induction hypothesis in which the assert is replaced with the assume by invoking $simpleAbs$ with flag set as $INDUCTION$, and then it outputs the induction step.

7) Loop termination assume: Finally, an assume is generated to constrain the values at this point such that the loop condition is violated (line 9 in Fig. 1g). This assumption for the outer loop in Fig. 1a will be unreachable as the loop is infinite.

The abstraction generated for sample code in Fig. 1a is shown in Fig. 1d, Fig. 1h and Fig. 1k and is successfully verified using bounded model checking. Fig. 1d contains an additional assumption on line 19 based on the structure of the input code that in each iteration either the if block or the corresponding $else$ block of the original code will be executed.

F. LABMC : Loop Abstraction for Bounded Model Checking

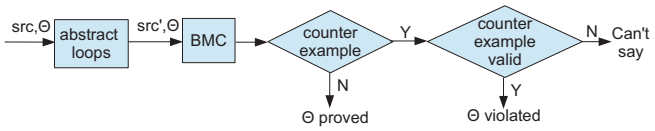

Fig. 3. Block Diagram of $LABMC$

Fig. 3 illustrates LABMC. Consider that a property Θ is to be verified in an input C program src. Θ is encoded as an assertion in src. LABMC abstracts all the loops in src by applying the algorithm explained in Section II-E to generate a transformed program src' of smaller known bounds. Then a BMC verifies src' with respect to the property Θ. If the verification does not result in a counterexample, then Θ is proved. Otherwise LABMC validates the counterexample by execution on src. If the counterexample is valid then Θ is violated otherwise LABMC cannot validate Θ to be proved or violated in src.

III. Experimentation

To evaluate the effectiveness of LABMC, we implemented it in a prototype tool and conducted two experiments on a 2.4 GHz Intel Xeon processor with 20 GB of RAM. In the first experiment we compared the performance of three best in class BMCs with LABMC over SV-COMP14 loop benchmarks. In the second experiment, we assessed the practical applicability of LABMC by evaluating it on an industrial automotive application. For all our experiments we assumed a time out value of ten minutes.

For our experiments we used CBMC version 4.7, ESBMC version 1.22.1 and LLBMC version 2013.1. We ensured their sound behaviour by providing them appropriate loop bound values for each input loop with a known bound. We provided the loop bound using the option $unwind\ \langle loopbound \rangle$ of CBMC and ESBMC and $max-loop-iterations = \langle loopbound \rangle$ of LLBMC, when $loopbound$ was known. When the bound of the input loop was not known or the loop was infinite, we did not provide any value. This ensured that when CBMC, ESBMC and LLBMC proved a property to be valid, it indeed was. For the transformed programs, while using BMC with LA we did not provide loop bounds to the BMC as all loops had known bounds.

A. Comparison between BMC and LABMC

As mentioned earlier, we used the SV-COMP14 loop benchmarks for this experiment. These benchmarks have been developed and maintained by the verification community and contain explicit properties to be verified. For our experiments we considered programs that did not contain operations on the contents of arrays and pointers in loops, as per the scope of LABMC. In the loop benchmarks, there are 19 such programs with valid properties (called safe programs) and 14 such programs with invalid properties (called unsafe programs).

In Table I rows 1-3 show the results of running each BMC over safe and unsafe programs. Rows 4-6 show the results of running loop abstraction (LA) followed by the respective BMC. The results of the three BMCs here are different from those presented in the SV-COMP14 competition because we ran them with a sound set of options as explained earlier. In SV-COMP14 the $max-loop-iterations = \langle loopbound \rangle$ option was used to provide a loop bound to LLBMC for loops of known and unknown bounds along with the option $no-max-loop-iterations-checks$ which ignores the checks on the sufficiency of the provided loop bound and thus may lead to unsound results. The corresponding options provided to ESBMC were $unwind\ \langle loopbound \rangle$ and $no-unwinding-assertion$. The options used for CBMC were not mentioned.

TABLE I
Experimental results on SV-COMP14 loop benchmark

Tool	19 Safe Programs			14 Unsafe Programs			
	S	TO	IA	No Bound		Bound 20	
				S	TO	S	IA
CBMC	2	17	0	2	12	14	0
ESBMC	2	17	0	1	13	11	3
LLBMC	2	0	17	6	8	11	3
LA+CBMC	17	0	2	14	0	-	-
LA+ESBMC	17	0	2	14	0	-	-
LA+LLBMC	13	0	6	14	0	-	-

S-Number of programs successfully verified, TO-Number of programs on which the BMC timed out, IA- Number of programs on which analysis was incomplete

TABLE II
Battery Controller Module (BCM) Information

Metrics	Values
Total loops in BCM	272
Sliced programs	186
Average relevant loops in a program	50
Average loops with unknown bound in a program	25
Average loops with bound greater than 100 in a program	5
Average loops with bound less than 100 in a program	20
Average loops reading array/pointer content in a program	1
Average loops abstracted in a program	29

Result Analysis - Safe Programs: As expected, each BMC performed better with LA than by itself on all programs. Of the 19 safe programs, all BMCs failed on a set of 17 programs. These programs contained loops with unknown bounds. So CBMC and ESBMC timed out during the loop unrolling phase whereas LLBMC exit with incomplete analysis. The reason being, for these programs to ensure a sound analysis we did not specify any loop unwinding bounds. On the other hand, as the abstracted programs had reduced bounds, CBMC and ESBMC were able to scale up on 17 programs after applying LA. Whereas LA+LLBMC continued to fail on 4 of those 17 programs due to some loop conditions being compound conditions. For two of the programs LABMC failed to prove the property (generated invalid counterexamples) with all BMCs due to over-approximation.

Result Analysis - Unsafe Programs: Among the 14 unsafe programs, 13 programs had loop(s) with an unknown bound. BMCs timed out in many programs when no bound was specified. However when we provided a sound bound of 20, all BMCs could generate valid counter examples in many cases. But for all programs with LA, all BMCs could generate valid counter examples without providing any bound. However, as LABMC generates an over-approximation of the original program, in practice, it is more suitable to prove properties and not for finding bugs.

TABLE III
COMPARISON WITH OTHER MODEL CHECKERS ON BCM

P	CBMC	LLBMC	ESBMC	SAT	IMP	BLAST	LA+CBMC
186	0	1	0	40	1	5	131

P-Number of programs, SAT-SATABS, IMP-IMPARA

B. LABMC on an automotive application

We performed this experiment on a real world Battery Controller Module (BCM) of a car for array index out of bounds property. The BCM has 186 array accesses. We first annotated the BCM module with an assert corresponding to each array access. Next, we sliced the annotated module relative to each of the assert and created 186 programs. Slicing ensured that all the loops present in it were relevant to the assert. The (average) number of different loops observed in these programs are presented in Table II. We applied LA+CBMC to each program as illustrated in Fig. 3. In the *abstract loops* phase, we abstracted all loops except those that contained operations on the contents of arrays or pointers, and loops with bounds less than 100. On an average, in each program we abstracted 29 loops out of 50. LA+CBMC could prove the safety of 131 programs. It timed out for the remaining 55 programs because they contained the original loops of unknown bounds with operations on the contents of arrays and pointers, therefore LA did not abstract them.

We also verified these 186 programs using CBMC, ESBMC and LLBMC, and other state of the art model checkers such as SATABS version 3.0, BLAST version 2.7.2 and IMPARA version 0.2. For the BMCs we did not specify any loop unwinding bound to ensure a sound analysis. The number of programs successfully verified by each tool are presented in the Table III. LA+CBMC outperformed all other model checkers. It proved more than 70% array accesses to be within array bounds thus showing the practical applicability of combining loop abstraction with bounded model checking. We manually verified and confirmed that these 131 array accesses were indeed within array bounds.

IV. RELATED WORK

The technique to over-approximate loops presented by Darke et. al in [7] does not yield a precise abstraction of output variables. In this paper, we attempt to improve the precision of output variables using acceleration and induction. The tool LAV [13] uses under-approximate and over-approximate unrolling of loops for finding errors and proving properties. Another technique presented in [9] by Kroening et. al under-approximates loops using acceleration to enable various verification engines including BMCs to find deep bugs in programs. Whereas our technique over-approximates loops using acceleration to enable BMCs to prove properties of programs. Abstract acceleration to abstract loops has also been presented by Halbwachs et al. and Schrammel et al. in [10] and [12], we can plug in those acceleration techniques to LABMC.

In [8] Donaldson et. al present K-induction for software verification. However, *K-Inductor*, the verifier which implements this approach could not prove the property in the example shown in Fig. 1a.

Additionally, the technique ABC [11] infers the exact upper bounds on the number of iterations of nested loops (for a special class of loops only) which may not be sufficiently small to scale up bounded model checking.

To the best of our knowledge, no one has combined the idea of induction and acceleration to over-approximate loops to prove properties and our experimental results indicate that this novel idea works well in practice.

V. CONCLUSION AND FUTURE WORK

Our over-approximation technique for loops has enabled BMCs to prove properties on applications containing loops of large or unknown bounds. This shows that combining abstractions such as output abstraction, abstract acceleration and induction help scale up bounded model checking. Going ahead, we wish to explore combining other suitable techniques to widen the scope of LABMC, for instance to handle loops that modify array contents. We also wish to extend the capability of LABMC to check for property violations by augmenting its abstraction procedure with a refinement procedure.

REFERENCES

[1] ESBMC 1.22. http://users.ecs.soton.ac.uk/lcc08r/esbmc/papers/TACAS13.pdf.

[2] LLBMC: Improved Bounded Model Checking of C Programs Using LLVM. http://llbmc.org/files/papers/TACAS13.pdf.

[3] SV-COMP 2014 Benchmark. http://sv-comp.sosy-lab.org/2014/benchmarks.php.

[4] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[5] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[6] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

[7] P. Darke, M. Khanzode, A. Nair, U. Shrotri, and R. Venkatesh. Precise analysis of large industry code. In *Asia Pacific Software Engineering Conferece*, pages 306–309, 2012.

[8] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Ruemmer. Software verification using k-induction. In *Proceedings of the 18th International Static Analysis Symposium (SAS'11)*, volume 6887 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2011.

[9] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*. Springer, 2013.

[10] Nicolas Halbwachs Laure Gonnord. Combining widening and acceleration in linear relation analysis. In *Static Analysis*, pages 144–160. Springer, 2006.

[11] Thibaud Hottelier Regis Blanc, Thomas Henzinger and Laura Kovacs. Abc: Algebraic bound computation for loops. In *Lecture Notes in Computer Science Volume 6355*, pages 103–118. Springer, 2010.

[12] Peter Schrammel and Bertrand Jeannet. Applying abstract acceleration to (co-)reachability analysis of reactive programs. *J. Symb. Comput.*, pages 1512–1532, 2012.

[13] Milena Vujosevic-Janicic and Viktor Kuncak. Development and evaluation of lav: An smt-based error finding platform - system description. In *VSTTE*, pages 98–113, 2012.

[14] Björn Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multi-threaded software with Impact. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 210–217, 2013.