0368.4435 Automatic Software Verification **April 28, 2015**

Lecture 6: Bounded Model Checking

Lecturer: Mooly Sagiv *Scribe: Amit Lichtenberg*

6.1 Overview

In this class we cover the area of Model Checking: checking weather a given model satisfies an interesting, predefined property. We will specifically focus on the problem of Bounded Model Checking, where the model is bounded in size, and its relation to software verification problems. We will then describe a specific implementation of a Bounded Model Checker - CBMC, which implements BMC over the C programming language, and discuss the challenges of implementing such a program and the solutions used by the CBMC authors to solve them.

6.2 Introduction

6.2.1 Model Checking

Model Checking describes a wide spectrum of problems, of the form: Does a given model, M, satisfies a property, P: $M \models P$. The model M is usually described by a state machine, while the property P is given by a logic formula over the machine states.

Some examples of problems which can be describes as a Model Checking problem:

- Does a given program ever dereference a null pointer? In this problem, the model M may describe the set of possible program states, and the property P will be True if at any state, a null pointer dereference may occur.
- Does the given car can ever overheat? The car too can be modeled by a state machine, while the property will check if at some state the car is overheated.
- Does a given circuit calculates the target function? The model will formalize the circuit states, and the property will check if at terminating states, the result is indeed the target function.
- Does a given program ever halts? This problem is infact the halting problem, which is known to be undecidable, but nevertheless it can be described as a model-checking problem and reasoned about using the model-checking techniques. There are many other model-checking problems which turn out to be hard, or undecidable, but to which we can still try to apply the same techniques.

The model M, as well as the state machine which describes it, can either be bounded or infinite. For example, one can describe a computer program P over variables $x_1, ..., x_n$, by defining a state machine S in which each state represent a possible combination of assignments for the variables. If the values are computationally bounded, e.g. 32 bit values, then the set of states is bounded as well by 2^{32n}. However, if the arithmetic model is unbounded, then so are the values of the variables, and so the set of possible states is infinite. Intuitively, verifying a property P over an infinite model is more complex then doing so over a bounded one. However, even if the given model is known to be bounded, and in some cases even if the bound itself is known (but is very large), the task of verifying a property is not always simple.

As some of the examples show, the problem of model checking is closely related with software verification: we represent the software as a (not necessarily finite) state machine, and formalize the verified property as a logic formula over the set of states. We would then like to prove that each traversal of the states preserves the property, or show a contradiction: a traversal of the states which shows a counterexample, by violating the property.

Before we continue, it is worth mentioning that the field of Model Checking handles a wide variety of other problems as well, and so we only cover the tip of the iceberg.

6.2.2 Bounded Model Checking

In the Bounded Model Checking (BMC) problem we treat the model, M, as a **finite** state machine with at most k states for some constant k. In other words, our problem is now: given a finite state machine M and a property P, either show a counter-example for P with at most k state transitions, or argue there is no such example.

The problem of BMC can be formulated as a SAT problem. To show how, we first provide a basic formalization for a state machine S which describes a model M. The state machine is composed of n states $s_1, ..., s_n$ and m transitions, where each transition φ_i is a triplet (s, α, s'): if the current state is s and the condition α holds, move to state s'. We assume for simplicity that no two α conditions exiting from one state coincide, i.e. the transitions are well defined. We additionally assume that there exists only a single starting state marked s_0. We also omit transitions which perform operations (e.g. assignments) - these can be added to the model quite simply, or just ignore by allowing dynamically defined states (e.g. simulate $x = x + 1$ using a new state where the new value of x is $+1$ then in the previous state).

Now, recall that we wish to find a k-path at the end of which P doesn't hold, or argue that no such path exists. In other words, we want to find a series of $k + 1$ states $s'_0, \ldots, s'_k$ such that:

- $s'_0 = s_0$ is the beginning state
- for each two adjacent states s'_i, s'_{i+1}, there exists a transition φ which moves between them.
- P does not host at s'_k

This can be formulated as a SAT formula in the following straightforward way:

$$\exists s'_0, ..., s'_k \in \{s_i : 1 \leq i \leq n\} : s'_0 = s_0 \wedge (\forall 0 \leq i < k : T(s'_i, s'_{i+1})) \wedge \neg P(s'_k) \tag{6.1}$$

Where $T(s, s')$ is true iff $\exists \varphi \in \{\varphi_i : 1 \leq i \leq m\} : \varphi = (s, \alpha, s')$ for some α.

If equation (6.1) is satisfiable, then a satisfying assignment of $s'_0, ..., s'_k$ is a straightforward counter-example for P. On the other hand, if it is not satisfiable, then there does not exist any k-path which breaks P.

We can also look at BMC as somewhat of an equivalent of bounded symbolic evaluation, which we have already seen in previous classes. As we recall, the symbolic evaluation technique constructed a state machine between different program states, where the transitions are program statements. Then, the symbolic evaluation traversed different paths, while checking if paths are feasible and lead to unwanted program states. When the symbolic execution is bounded in number of states, the problem becomes a BMC problem: check if the state machine which describes the problem contains an unwanted, feasible, state. For example, the unfolding of program loops upto k in symbolic execution is equivalent to limiting state machine exploration to about k steps.

6.3 CBMC

6.3.1 Introduction

CBMC [1] is a program developed at Carnegie Mellon University (CMU) by Daniel Kroening et al. It is a bounded model checker for C and C++ programs. As stated by the authors, it supports most C flavors, including C89, C99 anc most of C11, including most compiler extensions provided by common compilers (gcc and Visual Studio). While being mostly aimed towards embedded programs, it supports dynamic allocations using malloc and free, buffer overflow verification, pointer safety, exceptions and user-defined assertions. It is available on windows and linux. It is known to scale to programs with more then 30 thousand lines of code, and was used to find previously unknown bugs in Windows Device Drivers.

Apart from the basic application of verifying C programs, CBMC is used for a wide variety of applications, such as [2]:

- Error explanation: extended versions of CBMC can find and explain the cause of an error
- BMC of concurrent programs: verifying C programs executed by multiple threads
- Equivalence checking: checking weather two programs are equivalent, in the sense that they always compute the same output.
- Verifying embedded programs and non-program models: the models are formalized using C code and verified using CBMC
- Verifying existing programs, such as Linux and Windows device drivers
- Worst-case execution time: analyzing the execution time of programs
- Security: measuring information leakage in programs, finding security bugs in windows binaries and more

[1] C Bounded Model Checker, `http://www.cs.cmu.edu/~modelcheck/cbmc/`

[2] A more comprehensive list of CBMC applications and usage examples can be found at the Applications of CBMC page: `http://www.cprover.org/cbmc/applications.shtml`

6.3.2 How It Works

Generally, the CBMC can be described as a fairly simple program:

- The program is fed into the program analysis engine, along with a claim (property to satisfy) and a bound k, which defines the maximal "unfolding" done to program loops.
- The analysis engine then generates a CNF formula which describes the program, along with a term which describes the claim or property. Specifically, it looks for an assignment which satisfies both the problem and the **negation** of the claim, to show a contradiction of the claim, or prove that no such contradiction exists upto the given execution bound k.
- The resulting CNF is then fed into a SAT Solver, which either states it is satisfiable (ideally while showing a counterexample which disproves the claim), or argues that it is unsatisfiable, meaning the property is held.

Obviously, the implementation itself isn't as simple, and requires dealing with a lot of small issues, such as loops, syntactic sugar, arithmetic model, etc.

The main challenge in implementing a BMC for a program is in describing it using a set of Boolean conditions. We will now describe how CBMC does that for C programs, while focusing on main language structures and fairly simple examples. The next step will be to use almost any SAT Solver to construct a counter example, and then to translate that example back to the "program language" in a way that is useful for the programmer to actually solve the bug - this section will be described only briefly.

Control Flow Simplifications

In the first stage, the program is simplified in a way that removes all syntactic sugar and "convineiency" structures in the code. This is done in order to ease the work of the analyzer. For example:

- Side affects are removed:

  ```
  j = i++;
  ```

  is converted to:

  ```
  j = i;
  i = i + 1;
  ```

- Control flow keywords are made explicit by converting them to goto:

  ```
  while (1) {
    if (x == 200)
      break;
    x = x + 1;
  }
  y = x;
  ```

is converted to:

```
   while (1) {
      if (x == 200)
         goto while_exit;
      x = x + 1;
   }
while_exit:
   y = x;
```

- All loop forms are simplified into a single form, using ”while” loops:

  ```
  for (int i = 0; i < 10; i++)
     j = j * i;
  ```

  is converted to:

  ```
  i = 0;
  while (i < 10):
     j = j * i;
     i++;
  ```

By the end of this stage, all non-required structures and semantics have been removed from the programs. This allows CBMC to ignore those structures in the next stages, thus removing a lot of program complexities.

Loop Unwinding

In this stage, all program loops are unwound (unfolded) upto a limit k, which is given as an input parameter to the analysis. Recall that at this point the loops are only in form of ”while” loops, with possible ”goto”s used for control flows, thanks to the Control-Flow Simplification stage. We now unfold the loops in a trivial way:

```
while (cond) {
   Body;
}
Remainder;
```

1-unwinding:

```
if (cond) {
   Body;
   while (cond) {
      Body;
   }
}
Remainder;
```

After k-unwindings, the inner-most ”while” loop is either removed or replaced by an **unwinding assertion**. The unwinding assertion is used to verify that the unwinding was sufficient, i.e. that the loop cannot be executed more then k times (where k is the input bound). Specifically, supposed we used 2-unwinding of the above example. The result will be:

```
if (cond) { // first unwind
   Body;
   if (cond) { // second unwind
      Body;
      assert (!cond); // unwinding assertion
   }
}
Remainder;
```

This results in changes either in the soundness or in the correctness of the verification: Assume k was used as an unwinding bound, and the loop was infact executed $k' > k$ times, meaning the unwinding was insufficient. Then:

- If an unwinding assertion was used, an error will be wrongly reported: the loop was executed more then k times, but this does not necessarily mean that an error occurred.
- If an unwinding assertion was not used, and an error might happen in the $k+1$-th loop entry, then the error was not reported.

CBMC allows defining different unwind bounds for different program loops, allowing the analysis to be more efficient, but requiring more work from the programmer.

Converting to Single Static Assignment

At this point, we have a loop-free program (recall that loops were removed in the previous step), and we would like to convert it a set of Boolean formulas. The conversion is generally fairly simple: we would want to convert $x = y + 1; z = 3$ to a CNF equation of the form $x = y + 1 \wedge z = 3$. However, what happens when a variable is assigned more then once, i.e. $x = y + 1; x = x * 3$? Then the naive solution will result in the equation $x = y + 1 \wedge x = x \cdot 3$, which obviously does not capture the original intention. To solve this, we convert our program assignments to Single Static Assignments, meaning each variable is assigned only once. Practically, whenever a variable is assigned more then once, we allocate and use a new variable for each right-hand side of each new assignment:

Original Program:

```
x = x + y;
x = x * 2;
```

SSA-Program:

```
x1 = x0 + y0;
x2 = x1 * 2;
```

We will call these transformations of assignments ρ rules. The ρ rules are defined quite intuitively when it comes to basic assignments as in the example above, are not as intuitive in more complex examples:

- Conditionals: consider the following program:

  ```
  if (v)
     x = y;
  else
     x = z;
  w = x;
  ```

  According to the example we already saw, we would like to convert different assignments of x to different new variables, i.e.:

  ```
  if (v0)
     x0 = y0;
  else
     x1 = z0;
  w0 = ???;
  ```

  However, at the end of the condition, we do not know to which value of x w should be set. For this, we define a "conditional" operator and use it for each joint point (i.e. a point at which two possible values are possible for an assignment). In this case, the SSA-Program will look like this:

  ```
  if (v0)
     x0 = y0;
  else
     x1 = z0;
  x2 = v0 ? x0 : x1;
  w0 = x2;
  ```

- Unbounded arrays: consider an assignment of the form $A[1] = 5; A[2] = 10; A[k] = 20;$. To handle these assignments, we will treat each occurrence of A as a new variable Ai, similar to what done previously. The new variable Ai will be identical to the previous one $A(i-1)$ at every index but the one at which the assignment occurred, i.e.:

  Original Program:

  ```
  A[1] = 5;
  A[2] = 10;
  A[k] = 20;
  ```

  SSA-Program:

  ```
  A1 = lambda i: i == 1 ? 5 : A0[i];
  A2 = lambda i: i == 2 ? 10 : A1[i];
  ```

```
A3 = lambda i: i == k ? 20 : A2[i];
```

Note the new notation: $A = \text{lambda } i : i == v_0 : v_1 ? v_2$. The semantics of the notation is as follows: A is a function of variable i (the array index), where if i equals v_0, the result is v_1, and otherwise the result is v_2. Specifically in the above SSA-Program, A1 is a function of i, where if i equals 1 the output is 5, and otherwise the output is the value of (the function) $A0$ on input i, etc.

- Pointers: handling pointers is not simple, since the pointer space is practically limited only by memory size. The implementation is designed to be coherent with the semantics of C. Intuitively, we treat pointer assignment just like regular variable assignments using ρ rules. The challenge is when a pointer dereference occurs. In this case, we can use the case-split on the pointer value, similar to the if-then-else treatment.

A complete example for SSA-converting: Original Program:

```
int x, y;
y = 8;
if (x)
   y--;
else
   y++;

assert (y == 7 || y == 9);
```

SSA-Program:

```
int x, y;
y0 = 8;
if (x0)
   y1 = y0 - 1;
else
   y2 = y0 + 1;
y3 = x0 ? y1 : y2;

assert (y3 == 7 || y3 == 9);
```

Converting to CNF Formulas

Given that the program was converted to an SSA-Program in the previous stage, the conversion to CNF is simple. Consider the previous example:

```
int x, y;
y0 = 8;
if (x0)
   y1 = y0 - 1;
else
```

```
   y2 = y0 + 1;
y3 = x0 ? y1 : y2;

assert (y3 == 7 || y3 == 9);
```

The resulting CNF formula is as follows:

$$(y_0 = 8 \wedge y_1 = y_0 - 1 \wedge y_2 = y_0 + 1 \wedge y_3 = x_0?y_1 : y_2) \Rightarrow (y_3 = 7 \vee y_3 = 9) \tag{6.2}$$

To show that the assertion is violated, we will negate this formula, resulting in:

$$(y_0 = 8 \wedge y_1 = y_0 - 1 \wedge y_2 = y_0 + 1 \wedge y_3 = x_0?y_1 : y_2) \wedge \neg(y_3 = 7 \vee y_3 = 9) \tag{6.3}$$

The above formula can now be fed into our SAT solver.

Handling 32-bit Arithmetic

Unlike real-number arithmetic, numbers in C programs are limited to 32 bits. This poses a challenge when implementing the SAT Solver: in some ways, determining satisfiability over the limited model of 32-bit arithmetic is more complex then over unlimited models.

Still, CBMC solves this solution by modeling arithmetic using bit-vector arithmetic. This means each numeric value (int, double or float) is converted to its 32-bit vector representation. Arithmetic operators such as +, -, * and / are modeled using circuits. This results in non-efficient multiplication and division, but simplifies the model. Float / double types are modeled using fixed-point arithmetic (which limits the power of the BMC).

When handling bit-vector arithmetic, CBMC provides integration with a set of SAT solvers, some experimental and others more commonly used, such as Boolector, MathSAT , CVC3 , Yices and Z3.

6.3.3 Output Beautification

This all works quite well in practice, except that the results are often not easy to work with. The problem is that the SAT solver is designed to find some unsat assignment, but does not bother itself with finding a ”convinient” unsat assignment (which in itself isn’t even easy to define). Specifically, using the bit-vector arithmetic amplifies this problem, due to the ”unnatural” way of representing numbers.

Consider the following example (shown in class):

```c
void f(int a, int b, int c)
{
   int temp;
   if (a > b) {temp = a; a = b; b = temp;}
   if (b > c) {temp = b; b = c; c = temp;}
   if (a < b) {temp = a; a = b; b = temp;}
   assert (a<=b && b<=c);
}
```

Looking at this program, one can quite easily find an assignment which violates the assertion. Perhaps the simplest example would be $a = 1, b = 1, c = 0$. Running CBMC on this example results in the following output:

```
Counterexample:

State 17 file test.c line 3 thread 0
-----------------------------------------------------
a=603979795 (00100100000000000000000000010011)

State 18 file test.c line 3 thread 0
-----------------------------------------------------
b=603979795 (00100100000000000000000000010011)

State 19 file test.c line 3 thread 0
-----------------------------------------------------
c=603979794 (00100100000000000000000000010010)

State 20 file test.c line 5 function f thread 0
-----------------------------------------------------
temp=0 (00000000000000000000000000000000)

State 23 file test.c line 7 function f thread 0
-----------------------------------------------------
temp=603979795 (00100100000000000000000000010011)

State 24 file test.c line 7 function f thread 0
-----------------------------------------------------
b=603979794 (00100100000000000000000000010010)

State 25 file test.c line 7 function f thread 0
-----------------------------------------------------
c=603979795 (00100100000000000000000000010011)

Violated property:
file test.c line 9 function f
assertion a<=b && b<=c
FALSE

VERIFICATION FAILED
```

Indeed, a counterexample is found. However, the numbers used for the counterexample are enormous: $a = 603979795, b = 603979795, c = 603979794$. Taking another look, we can see that this is in fact exactly the same counterexample, where $a == b$ and $c == a + 1$. But just by looking at the numbers, this does not pop out. As the programmer and debugger of the program, we would much rather be provided with the simpler counter-example.

Another problem with the counter-examples returned by CBMC involves the execution path it takes. The execution path chosen by the SAT solver is usually not minimal, meaning the counter example might contain entering conditions which are not really required for the counter-example to work. Not only that, due to the way we chose to convert conditionals to SSA, we practically have a variable which indicates whether or not each "if" was taken, even if the "if" did not effect the final outcome. This too creates a lot of noise for the programmer.

So in summary, CBMC generates an output, but the output is not the optimal output to work with. This is nice, mainly because now we have more the explore and write articles about! We will introduce some ideas as per how to "prittify" the output, while focusing on the general concept and not getting into details.

Output Minimization and PBS

As we saw in one of the examples, in some cases we would like to be given the "smallest" possible counterexample. We will first need to define was "smallest" means in our case, and then think of how it can be implemented.

Suppose the SAT solver resulted in a counterexample which results in some execution path ϕ, which contains G transitions $\phi_i \to \phi_{i+1}$. We would like to minimize this path, in the sense of bringing it to a local minima which still results in a counter example, but where each removed transition results in a valid execution (and not a counter-example). We can formulate this as the following problem:

$$min \sum_{g \in G} l_g \cdot l_w \tag{6.4}$$

Where $l_g = 0/1$ indicates if the transition is taken, and l_w represents the weight of the transition, which can be taken to be the number of assignments in the transition.

Next, we would like to minimize the (absolute) value of variables, in order to find a "minimal value" counter-example. Formally, we would like to minimize $\sum x_i$, where x_i are the set of program variables.

These two problems relate to a general problem called Pseudo Boolean Solver (PBS). The problem is, given a set of CNF constraints over numeric values, and an addition set of optimization constraints given as $\sum_i w_i x_i$, output a decision (SAT or UNSAT). If the decision is SAT, we also output an satisfying assignment which is optimal given the optimization constraints.

There are some existing implementations for the PBS problem, some integrated into SAT solvers, such as MiniSat and PBS.

CBMC provides built-in greedy output beautification using similar techniques.

6.3.4 Handling External Code

The previous sections described how existing C code is converted by CBMC to easily analyzable code. However, the different simplification stages all assume that the code is available for analysis, namely it was written by the programmer and can be inputted to the analyzer. This leaves us with the problem of handling external code, such as system calls, external libraries, etc.

CBMC provides two paradigms for handling external code:

Nondeterministic Functions

CBMC provides a non-C primitives which models nondeterministic functions, i.e. functions which may return simultaneously many different values.

Usage:

```
xxx nondeter_xxx();
```

e.g.

```
int nondeter_int();
```

defines a nondeterministic function call which returns a value of type xxx (int).

The nondeterministic primitive are useful for modeling external input, unknown environment, library functions, and practically anything which may behave differently in different execution environments.

Modeling using nondeterministic primitives is quite convenient, but might prove to be expensive. In practice, all possible execution paths of the nondeterministic calls have to be extended and explored, even those not possible in practice.

Assume-Guarantee Reasoning

The assume-guarantee methodology is similar, in a way, to the deductive reasoning method which was introduced in previous class. CBMC allows assuming post-conditions and guaranteeing pre-conditions of both external and internal functions. This way, analysis of function calls can be done in a per-function way, significantly reducing the computational requirements of the analyzer.

In short, to determine if some function call, to function foo(), satisfies the requested property P, we split the reasoning to two parts:

- **Assuming** some (pre-defined) pre-condition C on the input state of foo(), and assuming P is held on entry, is P kept on exit from foo()?

- Whenever foo() is called, is the pre-condition C of foo() **guaranteed** on entry?

Given that both the assume and the guarantee conditions hold, the analysis is guaranteed to be correct. However, as we have seen in the deductive reasoning methods, this places a certain amount of extra work on the programmer, which is not always convenient.

When foo() is taken from external code, we can use the assume-guarantee reasoning by just assuming its pre and post-conditions, without verifying its implementation. This simplify the testing, but opens a hole for potential bugs which originate from programmer mistakes.

It sometimes makes sense to combine the two methods. For example, one can use nondeterministic functions to model system calls (which usually can result in a very large set of unexpected results), while using assume-guarantees for ensuring library functions as well as separating the code analysis to small, local, functions.

6.4 Conclusions

To summarize, we have seen how we can use the power of BMC can to reason about real-life programming languages, specifically C. The support for a real, fairly common, programing languages such as C is a big advantage of CBMC: it can be used to check a wide variety of existing programs, as well as for modeling complex systems such as hardware, circuits, etc, in a relatively simple way. However, CBMC does not support the complete set of C feature, which means not every program can be checked by CBMC. In addition, C in itself is considered an old, error-prone language, and is used less and less as new languages evolve.

Having tried CBMC on a number of fairly simple examples, I have found it to be a strong but problematic tool. Recalling the problematic example in section 6.3.3, after trying the beautification feature of CBMC, the resulting counterexample was still rather ugly: $a = -1073741825, b = 0, c = -2147483648$ (but then again, I guess beauty is in the eyes of the beholder). Looking at the bit-interpretation of the values this might make sense, but is still hard to explain to a tester. Additionally, the use of unwinding assertions, while nice, isn't always feasible. Defining the exact upper bound for the number of unwinds of a loop is impractical, and without it - the use of a loop assertion will forever cause erroneous bugs. The assume-guarantee mechanism goes back to the same issues we saw in previous tools demonstrated in class: it is strong, but requires extensive work from the programmer, which is usually infeasible on real-life programs.

Nevertheless, CBMC provides a powerful tool and a good example, and one can argue that the more the language is prone to bugs, the more BMC (and software verification is general) is helpful, which means C is somewhat ideal for our cause.

Bibliography

[1] *The CProver User Manual.* `http://www.cprover.org/cbmc/doc/manual.pdf`.

[2] E. M. Clarke O. Strichman Y. Zhu A. Biere, A. Cimatti. Bounded model checking. *Advances in Computers, Academic Press, 58*, 2003.

[3] M. Tautschnig D. Kroening. *Tools and Algorithms for the Construction and Analysis of Systems*, chapter CBMC - C Bounded Model Checker. Springer Berlin Heidelberg, 2014.

[4] A. Gurfinkel. Introduction to CBMC, November 2012. based on slides by D. Kroening, `http://www.cs.cmu.edu/~emc/15414-f12/lecture/cbmc.pptx`.