Software Testing and Validation

A.A. 2025/2026

Corso di Laurea in Informatica

Bounded Model Checking

Igor Melatti

Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Towards Bounded Model Checking

- Explicit and symbolic model checking are good, but many systems cannot be checked by neither
  - RAM and/or execution time are over soon
- Symbolic model checking directly makes use of boolean formulas through OBDDs
- What about using CNF, so that SAT solvers can be employed?
  - modern SAT solvers are pretty good in many practical instances
  - notwithstanding the SAT problem is of course still NP-complete

Towards Bounded Model Checking

- One big problem: computing quantization, AND, OR and negation of a CNF is not straightforward
  - especially because instances from Model Checking are HUGE
  - also checking equivalence of two CNF is not trivial, as CNF is not canonical
- However, if we set a limit k to the length of paths (counterexamples), then most of this is not needed any more
  - copy R for k times, with small adjustments
- This is actually *bug hunting*: if the result is PASS, then there is not an error within k steps
  - but there could be one at $k + 1$...
  - however, this is better than simple testing, as errors within k steps can be ruled out

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Bounded Model Checking of Safety Properties

- In Bounded Model Checking (BMC) we are given a KS $\mathcal{S} = \langle S, I, R, L\rangle$, an LTL formula φ, and $k \in \mathbb{N}$ (also called *horizon*)
- Let us consider the LTL property $\varphi = \mathbf{G}p$, being $p \in AP$
- We want to find counterexamples (if any) of length exactly k
- If $x = x_1, \ldots, x_n$ with $n = \lceil \log_2 |S| \rceil$, let us consider $x^{(0)}, \ldots, x^{(k)}$
- $\mathcal{S} \models_k \mathbf{G}p$ iff the following CNF is unsatisfiable:

$$I(x^{(0)}) \wedge \bigwedge_{i=0}^{k-1} R(x^{(i)}, x^{(i+1)}) \wedge \neg p(x^{(k)})$$

  - otherwise, a satisfying assignment is a counterexample

Bounded Model Checking of Safety Properties

- Note that each $x^{(i)}$ encloses n boolean variables, thus we have $n(k+1)$ boolean variables in our SAT instance
  - the longest our horizon, the biggest our SAT instance
- Note that I and R must be in CNF, which is not difficult
  - NuSMV does this pretty well
- It is straightforward to modify the previous formula to detect counterexamples of length *at most* k
- However, it is usually preferred to perform BMC with increasing values for k
  - practically, till when the SAT solver goes out of computational resources
  - some approaches exist to estimate the *diameter* of a KS...

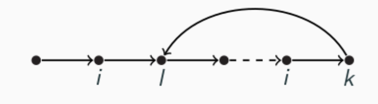
LTL Bounded Model Checking

- In order to perform BMC of a generic LTL property, we need to introduce *LTL bounded semantics*
  - $\mathcal{S} \models_k \varphi$ iff $\forall \pi \in \mathrm{Path}(\mathcal{S}).\ \pi \models_k \varphi$
  - so that, for any k, $\mathcal{S} \models_k \varphi$ implies $\mathcal{S} \models \varphi$
- For a given π, $\pi \models_k \varphi$ iff $\pi, 0 \models_k \varphi$, which is usually re-written as $\pi \models_k^0 \varphi$
- For a given π, we only consider $\pi|_k$; then, either $\pi|_k$ contains a self loop (i.e., it is *lasso-shaped*) or not
  - recall that $\pi|_k$ contains k transitions and $k+1$ states
- π is lasso-shaped iff $\pi = \rho\sigma^\omega$
  - there exists $l \leq k$ s.t. $\rho = \pi|_{l-1}$ and $\sigma = \pi(l) \dots \pi(k)$
  - ρ is empty for $l = 0$
  - π is a (k, l)-loop (more generally, a k-loop)
  - of course, $R(\pi(k), \pi(l))$ must hold

LTL Bounded Semantics for $\pi \models_k^i \varphi$

- Let $L(\pi, l, k)$ hold iff π has a (k, l)-lasso
- Let $L(\pi, k)$ hold iff π has a (k, l)-lasso for some $l \leq k$
- If $L(\pi, k)$ holds, we may consider $\pi(i)$ for $i > k$
  - this is possible because we know π to have a lasso
  - namely, if $L(\pi, l, k)$ holds, then

$$succ(i) = \begin{cases} i+1 & \text{if } i < k \\ (i \bmod k) + l & \text{otherwise} \end{cases}$$



LTL Bounded Semantics for $\pi \models_k^i \varphi$

- $\forall \pi \in \mathrm{Path}(\mathcal{S}), i \leq k.\ \pi \models_k^i \mathrm{true}$
- $\pi \models_k^i p$ iff $p \in L(\pi(i))$
- $\pi \models_k^i \Phi_1 \wedge \Phi_2$ iff $\pi \models_k^i \Phi_1 \wedge \pi \models_k^i \Phi_2$
- $\pi \models_k^i \neg\Phi$ iff $\pi \not\models_k^i \Phi$
- $\pi \models_k^i \mathbf{X}\Phi = \begin{cases} \pi \models_k^{i+1} \Phi & if\, L(\pi, k) \\ i < k \wedge \pi \models_k^{i+1} \Phi & \text{otherwise} \end{cases}$
- $\pi \models_k^i \Phi_1 \;\mathbf{U}\; \Phi_2 =$

$$\begin{cases} \exists m \geq i :\ \pi \models_k^m \Phi_2 \wedge \forall i \leq j < m.\ \pi \models_k^j \Phi_1 & if\, L(\pi, k) \\ \exists i \leq m \leq k :\ \pi \models_k^m \Phi_2 \wedge \forall i \leq j < m.\ \pi \models_k^j \Phi_1 & \text{otherwise} \end{cases}$$

LTL Bounded Semantics for $\pi \models^i_k \varphi$

- $\pi \models^i_k \mathbf{G}\Phi = \begin{cases} \forall j \geq i.\ \pi \models^j_k \Phi & \text{if } L(\pi, k) \\ \mathit{ff} & \text{otherwise} \end{cases}$
- $\pi \models^i_k \mathbf{F}\Phi = \begin{cases} \exists j \geq i.\ \pi \models^j_k \Phi & \text{if } L(\pi, k) \\ \exists i \leq j \leq k.\ \pi \models^j_k \Phi & \text{otherwise} \end{cases}$
  - note that $\mathbf{G}p \not\equiv \neg(\mathbf{F}\neg p)$ with bounded semantics!

Bounded Model Checking of LTL Properties

- Similarly to safety properties, for an LTL formula φ, $\mathcal{S} \models_k \varphi$ iff the following formula is unsatisfiable:

$$I(x^{(0)}) \wedge \bigwedge_{i=0}^{k-1} R(x^{(i)}, x^{(i+1)}) \wedge ((\neg L(k) \wedge [\![\neg\varphi]\!]_k^0) \vee (\bigvee_{l=0}^{k} L(l,k) \wedge [\![\neg\varphi]\!]_{k,l}^0))$$

  - to be translated into a CNF before being passed to a SAT solver
- $L(l, k)$ and $L(k)$ do not depend on a π: they represent the *possibility* that a path is a lasso
- Thus, $L(l, k) = R(x^{(k)}, x^{(l)})$ and $L(k) = \bigvee_{l=0}^{k} L(l, k)$

Bounded Model Checking of LTL Properties

- Formula is unsatisfiable for SAT solver:

$$I(x^{(0)}) \wedge \bigwedge_{i=0}^{k-1} R(x^{(i)}, x^{(i+1)}) \wedge ((\neg L(k) \wedge [\![\neg\varphi]\!]_k^0) \vee (\bigvee_{l=0}^{k} L(l,k) \wedge [\![\neg\varphi]\!]_{k,l}^0))$$

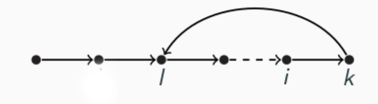
- We now have to define $[\![\varphi]\!]_k^0, [\![\varphi]\!]_{k,l}^0$
  - $[\![\varphi]\!]_k^0$ is in AND with $\neg L(k)$, thus for lasso-free path
  - $[\![\varphi]\!]_{k,l}^0$ is in AND with $L(k)$, thus for (k, l)-loops
- So that LTL bounded semantics is retained
  - for lasso shaped cases, we may look at what is before i when translating $[\![\varphi]\!]_k^i$

Bounded Model Checking of LTL Properties

- $[\![\mathrm{true}]\!]_k^i = [\![\mathrm{true}]\!]_{k,l}^i = tt$
- $[\![p]\!]_k^i = [\![p]\!]_{k,l}^i = p(x^{(i)})$
- $[\![\neg\Phi]\!]_k^i = \neg[\![\Phi]\!]_k^i$
- $[\![\neg\Phi]\!]_{k,l}^i = \neg[\![\Phi]\!]_{k,l}^i$
- $[\![\Phi_1 \wedge \Phi_2]\!]_k^i = [\![\Phi_1]\!]_k^i \wedge [\![\Phi_2]\!]_k^i$
- $[\![\Phi_1 \wedge \Phi_2]\!]_{k,l}^i = [\![\Phi_1]\!]_{k,l}^i \wedge [\![\Phi_2]\!]_{k,l}^i$
- $[\![\mathbf{X}\Phi]\!]_k^i = \begin{cases} [\![\Phi]\!]_k^{i+1} & \text{if } i < k \\ ff & \text{otherwise} \end{cases}$
- $[\![\mathbf{X}\Phi]\!]_{k,l}^i = [\![\Phi]\!]_{k,l}^{succ(i)}$

Bounded Model Checking of LTL Properties

- $[\![\Phi_1 \mathbf{U} \Phi_2]\!]_k^i = \bigvee_{j=i}^{k}([\![\Phi_2]\!]_k^j \wedge \bigwedge_{m=i}^{j-1}[\![\Phi_1]\!]_k^m)$
  - recall that $\exists$ is OR and $\forall$ is AND...
- $[\![\Phi_1 \mathbf{U} \Phi_2]\!]_{k,l}^i = \bigvee_{j=i}^{k}([\![\Phi_2]\!]_{k,l}^j \wedge \bigwedge_{m=i}^{j-1}[\![\Phi_1]\!]_{k,l}^m) \vee$
  $\bigvee_{j=l}^{i-1}([\![\Phi_2]\!]_{k,l}^j \wedge \bigwedge_{m=i}^{k}[\![\Phi_1]\!]_{k,l}^m \wedge \bigwedge_{m=l}^{j-1}[\![\Phi_1]\!]_{k,l}^m)$
  - note that the second big OR is not empty only if $l \leq i-1$, i.e., if the loop starts *before* i
  - thus, it deals with the case in which we have to "imagine" the infinite path
    - for the lasso-shaped case, bounded and unbounded semantics must be equivalent
  - essentially, it also adds the case in which Φ_2 does not hold from i to k, but it holds before, in the loop part
  - of course, Φ_1 must hold from i to k and till Φ_2



Bounded Model Checking of LTL Properties

- $[\![\mathbf{G}\Phi]\!]_k^i = ff$
  - "globally" cannot be guaranteed without loops!
- $[\![\mathbf{G}\Phi]\!]_{k,l}^i = \bigwedge_{j=\min\{i,l\}}^k [\![\Phi]\!]_{k,l}^j$
  - but if we have a loop, it is sufficient to have Φ globally inside the loop
  - plus the prefix, if any
- $[\![\mathbf{F}\Phi]\!]_k^i = \bigvee_{j=i}^k [\![\Phi]\!]_k^j$
- $[\![\mathbf{F}\Phi]\!]_{k,l}^i = \bigvee_{j=\min\{i,l\}}^k [\![\Phi]\!]_{k,l}^j$
  - no problem for "eventually"
- Also **R** should be given, no more expressible using **U**

Bounded Model Checking in NuSMV

- The following sequence is as before:
  1. `read_model`
  2. `flatten_hierarchy`
  3. `encode_variables`
- Then, `build_boolean_model` instead of `build_flat_model`
  - it uses a different representation, better suited for BMC
- Then, `bmc_setup` instead of `build_model`
  - instead of creating OBDDs, computes $I(x^{(0)}) \wedge R(x^{(0)}, x^{(1)})$, ready to be unfolded

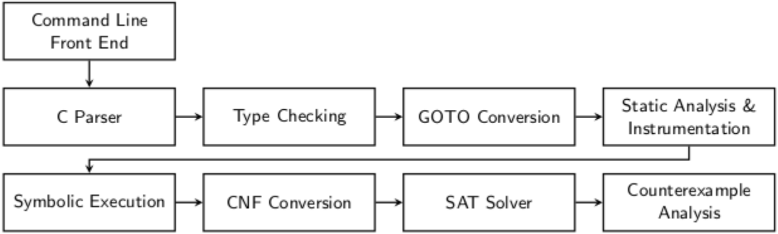
Bounded Model Checking in NuSMV

- Finally, `check_ltlspec_bmc -k` k
  - for k times, creates the input for SAT and invokes the SAT solver
  - if an error is found, it may stop before k
  - option `-o` of `check_ltlspec_bmc` also dumps the SAT instance in DIMACS format
  - this also entails negating the given LTL formula
  - "corrected", more intuitive semantics: $\mathbf{G}p$ is true on a non-lasso π if $\forall 1 \leq i \leq k . p(\pi(i))$ holds

Bounded Model Checking of Programs

- Till now, we had to write a model of the system under verification (SUV)
- There are some cases in which we can use the actual SUV, with little or no instrumentation
  - it is possible to translate a digital circuit to a NuSMV specification in a completely automated way (not difficult to imagine how...)
  - here, we want to deal with a rather surprising application of BMC: model checking a C program!
- CBMC is a model checker performing BMC of C programs with little or no instrumentation
  - thus, the input for CBMC is a C program (possibly with some added statements)
  - an integer k may be required too
  - again, output is PASS or FAIL (with a counterexample)
- We now give the main ideas of how it works

CBMC


Command Line Front End
C Parser
Type Checking
GOTO Conversion
Static Analysis & Instrumentation
Symbolic Execution
CNF Conversion
SAT Solver
Counterexample Analysis

CBMC

CLI Front End No GUI, you have to invoke CBMC from a shell
- one mandatory argument: the C file
- `-h` or `--help` for a complete list of options

C Parser the standard system parser, e.g., `gcc`
- this includes the preprocessor for `define` and other macros

Type Checking
- for all symbols (constants, variables and functions), keep track of the corresponding types
- including the number of bits needed

CBMC

GOTO Conversion for our purposes, we skip this

- used to optimize the symbolic execution part on loops

Static Analysis & Instrumentation resolve function pointers

- replaced with a case over all possible functions
- as a result, we have a static call graph
- generally speaking, static analysis is a further methodology for software verification
- hybrid between model checking and proof checkers
- here it is used in a lightweight way
- instrumentation: some assertions for invalid pointer operations and memory leaks are automatically added

CBMC: Symbolic Execution

- It is composed of two parts: loop unwinding and Static Single Assignment (SSA) form
- An additional parameter k is needed as the *unwinding number*
  - CBMC may also try some heuristics to guess the maximum unwinding for each loop
- If many loops are present, it is possible to set different unwinding numbers for each loop
- The unwinding number is usually interpreted as mandatory: an `assert` is added at the end
- It is possible to avoid this with option `--partial-loops`

CBMC: Loop Unwinding with $k = 3$

```
while (x <= 4) {
  y += f(3);
  x++;
}
```

```
if (x <= 4) {
  y += f(3);
  x++;
  if (x <= 4) {
    y += f(3);
    x++;
    if (x <= 4) {
      y += f(3);
      x++;
      /* with --partial-loops
         this is not added */
      assert(!(x <= 4));
    }
  }
}
```

CBMC: SSA

- Each assignment is treated separately, generating a copy of the left side
- If we only have n assignments, then n is our bound for BMC
  - if such assignments are inside a loop with unwinding k, then the size is kn
  - generally speaking, you have to sum on all loops and all loop-free assignments

```
x=x+y;
if(x!=1)
   x=2;
else
   x++;

assert(x<=3);
```

$\rightarrow$

```
x1=x0+y0;
if(x1!=1)
   x2=2;
else
   x3=x1+1;

x4=(x1!=1)?x2:x3;
assert(x4<=3);
```

CBMC: SSA and Pointers

- What abount pointers? e.g.,

```
int *p = malloc(n*sizeof(int));
p[n] = 0;
p[0] = 1;
```

- They become functions:
  $\lambda x.\ 0 \text{ if } x = n \text{ else } (1 \text{ if } x = 0 \text{ else } \perp)$
- Then, it is similar to the assignment on x_4 in the previous slide

CBMC: CNF Convertion

- The idea is again to have a CNF
  $I(x^{(0)}) \wedge \bigwedge_{i=0}^{k-1} R(x^{(i)}, x^{(i+1)}) \wedge \bigwedge_{i=a} \neg p_i(x^{(\alpha(i))})$
  - a is the number of assertions, and $\alpha(i)$ tells on which variables is defined the i-th assertion
  - of course, digital circuit logics (and ITE...) have to be used

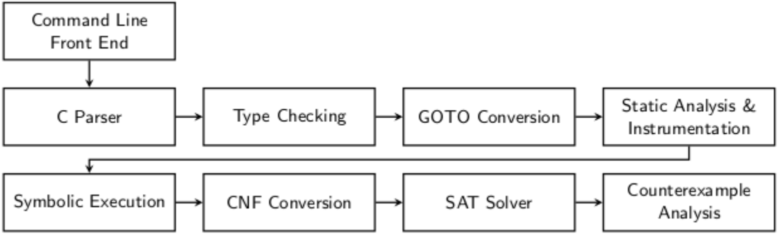
```
x1=x0+y0;
if(x1!=1)
   x2=2;
else
   x3=x1+1;

x4=(x1!=1)?x2:x3;
assert(x4<=3);
```

$\rightarrow$

$$
\begin{aligned}
C := {} & x_1 = x_0 + y_0 \ \wedge \\
& x_2 = 2 \ \wedge \\
& x_3 = x_1 + 1 \ \wedge \\
& x_4 = (x_1 \mathrel{!=} 1) ? x_2 : x_3 \\
P := {} & x_4 \leq 3
\end{aligned}
$$

CBMC


Command Line
Front End
C Parser
Type Checking
GOTO Conversion
Static Analysis &
Instrumentation
Symbolic Execution
CNF Conversion
SAT Solver
Counterexample
Analysis