The Bounded Model Checker LLBMC

**LLBMC**
- Bounded model checker for C programs
- Developed at KIT
- Successful in SV-COMP competitions

**Functionality**
- Integer overflow, division by zero, invalid bit shift
- Illegal memory access (array index out of bound, illegal pointer access, etc.)
- Invalid free, double free
- User-customizable checks (via __llbmc_assume / __llbmc_assert)

**Employed techniques**
- Loop unrolling, function inlining; LLVM as intermediate language
- SMT solvers, various optimizations (e.g. for handling array-lambda-expressions)
Model Checking Problem

- Consider a finite-state transition system $M = (S, I, T)$, where
  - $S$ is a set of states,
  - $I \subseteq S$ is the set of initial states and
  - $T \subseteq S \times S$ is a transition relation between states.
- A run of $M$ is a (finite or infinite) sequence $(s_1, s_2, \ldots, s_n, \ldots)$ of states such that $s_1 \in I$ and $(s_i, s_{i+1}) \in T$ for all $i \geq 1$.
- Let $B \subseteq S$ be a set of bad states.
- **Question**: Is there a run of $M$ which reaches a bad state? (i.e.: Is there a run with $s_i \in B$ for some $i$?)
- **Example**:
Model Checking Problem

• The model checking problem, as presented, is a graph reachability problem, and thus in principle easily solvable (explicit state model checking).

• However, the graph can be extremely large ($10^{1000}$ or more elements in state space)

• Moreover, the state space is typically structured:

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>Flip-flops, registers, ...</td>
</tr>
<tr>
<td>State space</td>
<td>Cartesian product of Boolean variables</td>
</tr>
<tr>
<td>Transitions</td>
<td>Updates to registers / flip-flops</td>
</tr>
</tbody>
</table>

• Thus, a symbolic representation of state space and transition relation can be used (and is typically much more efficient) => symbolic model checking
Symbolic Model Checking

• **Idea:** Use formulas to represent state sets and the transition relation.

• **Examples:**

**Hardware:**
• 2-bit counter going from 0 to 2, starting at 1
• State encoded in two latches b and c (b for the high-bit)
• Predicates for initial and bad states, transition relation:
  • I(s) = (¬b ∧ c), B(s) = (b ∧ c)
  • T(s,s') = (b' ⇔ c) ∧ (c' ⇔ ¬(b ∨ c))

**Software:**
• [0] int x=0;
  [1] while (x<4) {
    [2]   x++;
    [3] }
  [4] return x;

• State encoded as one integer and a program counter
• Predicates for I, B and T:
  • I(s) = (x=0 ∧ PC=0)
  • B(S) = (x>5)
  • T(s,s') = (PC=0 ⇒ x'=0 ∧ PC'=1) ∧ (PC=1 ∧ x<4 ⇒ PC'=2 ∧ x'=x) ∧ (PC=1 ∧ x>=4 ⇒ PC'=4 ∧ x'=x) ∧ (PC=2 ⇒ ∧ PC'=3 ∧ x'=x+1) ∧ ...
Symbolic Model Checking

• To check, whether a bad state is reachable, we need the transitive closure $T^*$ of $T$.
  • There is an error, if $I(s) \land T^*(s, s') \land B(s')$ is satisfiable.
• The transitive closure can be computed via a fixedpoint iteration.
• In the propositional case, BDDs (binary decision diagrams) are often used for representing $I$, $T$, $B$ and for computing the fixpoint.
Hardware Bounded Model Checking

- **Idea:** avoid computation of transitive closure / fixpoint
- Use prefixes of length k for checking paths (runs).

  - If
    \[
    \text{BMC}_k : \quad I(s_1) \land \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \land \bigvee_{i=1}^k B(s_i)
    \]
    is satisfiable, then there is a path of length k leading to a bad state.

  **Advantages:**
  - No need to compute transitive closure of T
  - Formula BMC$_k$ doesn't refer to a notion of state, can be solved with a SAT solver (if state variables are Boolean)

  **Disadvantages:**
  - k copies of state variables needed
  - Complete only if bound is sufficient (how do we know that?)
Software Bounded Model Checking

• We can use the same idea as for hardware for software.

• But there are also different, more efficient encodings, e.g.:
  • If a program is in **SSA form, contains no loops and function calls**, then each **variable is assigned in the whole program at most once**.
  • Thus, each assignment can be seen (and encoded) as a logical equality.

• We thus can use an encoding as follows (e.g., for an LLVM module):
  • For each instruction \( I \) (and successor instruction \( I' \)):
    \[
    c_{exec}(I) \Rightarrow \neg \text{Err}(I) \land \text{Enc}(I)
    \]
    \[
    c_{exec}(I') = c_{exec}(I) \land c_{branch}(I, I')
    \]
  • Here:
    • \( c_{exec} \) is the execution condition of instruction \( I \)
    • \( c_{branch} \) is the condition when control flow goes from \( I \) to \( I' \)
    • \( \text{Enc}(I) \) is the encoding of the effects of \( I \), \( \text{Err}(I) \) if there is an error executing instruction \( I \).
Alternative: Horn-Clause Encoding

- Use predicates \( Li(x,...) \) for program locations.
- Then write each program transition as a rule like, e.g.,
  - \( L1(x,y) \land x>5 \land y<10 \Rightarrow L2(x+1,y) \)
- Similar techniques are used in the Swift intermediate representation (SIL):

  In SIL, basic blocks take arguments, which are used as an alternative to LLVM’s phi nodes. Basic block arguments are bound by the branch from the predecessor block:

  ```
  sil @if : $(Builtin.Int1, Builtin.Int64, Builtin.Int64) -> Builtin.Int64 {
    bb0(%cond : $Builtin.Int1, %ifTrue : $Builtin.Int64, %ifFalse : $Builtin.Int64):
      cond_br %cond : $Builtin.Int1, then, else
      then: .
        br finish(%ifTrue : $Builtin.Int64)
    else:
      br finish(%ifFalse : $Builtin.Int64)
    finish(%result : $Builtin.Int64):
      return %result : $Builtin.Int64
  }
  ```
LLBMC Encoding: Basic Blocks

\[ c_{exec}(\text{entry}) = \text{true} \]
LLBMC Encoding: Basic Blocks

\[ c_{\text{exec}}(BB_2) = c_{\text{exec}}(BB_1) \]
LLBMC Encoding: Basic Blocks

\[
c_{\text{exec}}(BB_2) = c_{\text{exec}}(BB_1) \land c
\]

\[
c_{\text{exec}}(BB_3) = c_{\text{exec}}(BB_1) \land \neg c
\]
LLBMC Encoding: Basic Blocks

\[ c_{\text{exec}}(BB_3) = c_{\text{exec}}(BB_1) \land c_1 \lor c_{\text{exec}}(BB_2) \land c_2 \]
LLBMC Encoding: Instructions

```c
int f(int x, int y) {
    return ((x - y > 0) == (x > y));
}
```

```c
define i32 @f(i32 %x, i32 %y) {
    entry:
    %sub = sub nsw i32 %x, %y
    %cmp = icmp sgt i32 %sub, 0
    %conv = zext i1 %cmp to i32
    %cmp1 = icmp sgt i32 %x, %y
    %conv2 = zext i1 %cmp1 to i32
    %cmp3 = icmp eq i32 %conv, %conv2
    %conv4 = zext i1 %cmp3 to i32
    ret %conv4
}
```

\[
\begin{align*}
sub &= \text{bvsub } x \ y \\
\land \ cmp &= (\text{bvsigt } sub \ \text{bv}32,0) \ ? \ \text{bv}1,1 \ : \ \text{bv}1,0 \\
\land \ conv &= \text{zero\_extend}_{31} \ cmp \\
\land \ cmp1 &= (\text{bvsigt } x \ y) \ ? \ \text{bv}1,1 \ : \ \text{bv}1,0 \\
\land \ conv2 &= \text{zero\_extend}_{31} \ cmp1 \\
\land \ cmp3 &= (\text{conv} = \text{conv2}) \ ? \ \text{bv}1,1 \ : \ \text{bv}1,0 \\
\land \ conv4 &= \text{zero\_extend}_{31} \ cmp3
\end{align*}
\]
LLBMC Encoding: Phi Nodes

\[ x_2 = c_{\text{exec}}(BB_1) \land x_0 : x_1 \]
LLBMC Encoding: Loops
LLBMC Encoding: Memory Accesses

write : $A \times I \times E \rightarrow A$
read : $A \times I \rightarrow E$

\[
\text{read}(\text{write}(\text{write}(a_0, i_0, e_0), i_1, e_1), i_2, e_2), i_3)
\]
LLBMC Encoding: Heap State

$h_0 = \varepsilon$

$h_1 = \text{malloc}$, $h_0$, $p_0$, $s_0$

$h_2 = \text{malloc}$, $h_1$, $p_1$, $s_1$

$h_3 = \text{free}$, $h_2$, $p_2$

$b = \text{validaccess}$, $h_3$, $p_3$, $s_3$
Backwards Slicing

```c
int a, b;

int foo(int x, int y)
{
    int r = a, t = b;
    if (a > b) {
        t = a*2;
    }

    while (t > a) {
        t -= 2;
        y++;
    }

    if (x != 0) {
        b = x-a; // slice here
    } else {
        b = t+y;
    }

    return x+b;
}
```
Control Dependence Graph (CDG)

CFG for 'BINARYSEARCH_S16_Near_iL' function
LLBMC Command Line Options

$ llbmc --help
OVERVIEW: llbmc

USAGE: llbmc [options] <input bitcode files>

OPTIONS:
-arguments=<string> - Arguments to be passed to "main"
-fp-div-by-zero-checks - Check for floating-point division by zero
-fp-nan-checks - Check for NaN production in floating-point arithmetic
-function-name=<string> - Name of the function to be checked
-heap-model - Set the heap model:
  =eager - eager expansion as in SSV 2010 (default)
  =lazy - lazy expansion as in SMT 2011
-help - Display available options (-help-hidden for more)
-ignore-volatile - Treat volatile loads like non-volatile loads
-incremental - Incremental SMT solving (experimental)
-leak-check - Check for memory leaks
-log-level - Set log level to one of the following:
  =off - log nothing
  =error - log only errors
  =sparse - log on sparse level (default)
  =verbose - log on verbose level
  =debug - log on debug level
-mallocs-may-fail - Malloes may fail (i.e., return NULL)
-max-builtins-iterations=<uint> - Maximum number of times the loops in C library functions are executed
-max-function-call-depth=<uint> - Maximum number of function inlining steps
-max-loop-iterations=<uint> - Maximum number of times a loop is executed
-max-memcpy-iterations=<uint> - Maximum number of times the loops in memcpy/memmove/memset are executed
-memcpy - Set treatment for memcpy/memmove/memset:
  =instantiation-based - instantiation-based encoding (default)
  =eager - eager encoding
  =unroll - unroll loop
LLBMC Command Line Options

- **-no-custom-assertions**: Do not check custom assertions (`__llbmc_assert`)
- **-no-div-by-zero-checks**: Do not check for divisions by zero
- **-no-max-function-call-depth-checks**: Do not add assertions but assumptions for function calls
- **-no-max-loop-iterations-checks**: Do not add assertions but assumptions for backedges
- **-no-memcpy-disjoint-checks**: Do not check for disjointness of memory regions for `memcpy`
- **-no-memory-access-checks**: Do not check load and store operations
- **-no-memory-allocation-checks**: Do not check heap and stack allocation operations
- **-no-memory-free-checks**: Do not check free operations
- **-no-overflow-checks**: Do not check for signed overflows
- **-no-shift-checks**: Do not check bit shifts for too large shifts
- **-only-custom-assertions**: Only check custom assertions (`assert/__llbmc_assert`)
- **-output-file=<string>**: Output file name (if not stdout)
- **-smt-solver**: Set the SMT solver to use as a backend:
  - `=boolector` - Boolector with Lingeling
  - `=boolector-lambda-toasc` - Boolector with lambdarized ToASC and Lingeling
  - `=stp` - STP with MiniSat (default)
  - `=stp-msp` - STP with MiniSat and propagators
  - `=stp-sms` - STP with simplifying MiniSat
  - `=reference-count-debugger` - debug reference counting of SMT expressions
  - `=reference-count-debugger-lia` - debug reference counting of SMT expressions with LIA for bitvectors
- **-smt-solver-timeout=<int>**: Timeout (in seconds) for the SMT solver
- **-stack-promotion**: Set extent to which stack memory locations are promoted to registers:
  - `=on` - all promotable stack memory locations (default)
  - `=safe` - only promotable stack memory locations that are initialized in an obvious way
  - `=safe-expensive` - only promotable stack memory locations that are initialized (expensive check)
  - `=off` - no stack memory locations
- **-start-with-empty-heap**: Start without any allocations on the heap (default) (experimental if disabled)
LLBMC Command Line Options

Output options
- `result` - bug checking result
- `synopsis` - error synopsis
- `location` - error location
- `stacktrace` - LLVM stack trace
- `counterexample` - LLVM counter-example
- `bitcode` - LLVM bitcode (after transformations)
- `simple` - ILR (simple)
- `pretty` - ILR (pretty)
- `latex` - ILR (latex)
- `VCs` - ILR verification conditions
- `statistics` - ILR formula statistics
- `model` - ILR model
- `assertion` - ILR assertion information
- `variables` - ILR variable assignments
- `graphviz` - DOT format (Graphviz)
- `btor` - BTOR format (Boolector)
- `smtlib` - SMTLIB format
- `smtlib-uf` - SMTLIB format (using UFs)
- `smtlib2` - SMTLIB2 format (using "let")
- `smtlib2-uf` - SMTLIB2 format (using "let" and UFs)
- `smtlib2x` - SMTLIB2 format (using "define-fun")
- `smtlib2x-uf` - SMTLIB2 format (using "define-fun" and UFs)
- `smtlib2x-lia` - SMTLIB2 format (using "define-fun") using LIA for bitvectors
- `smtlib2x-uf-lia` - SMTLIB2 format (using "define-fun" and UFs) using LIA for bitvectors
- `stp-api` - C program calling STP's API
- `uninitialized-globals` - Do not initialize global variables
- `version` - Display the version of this program