Bounded Model Checking of Software for Real-World Applications

Parts 4-6

UniGR Summer School on Verification Technology, Systems & Applications
VTSA 2018
Nancy, France

Carsten Sinz
Institute for Theoretical Informatics (ITI)
Karlsruhe Institute of Technology (KIT)

30.08.2018

The Bounded Model Checker LLBMC

Karlsruhe Institute of Technology

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

```
struct list_node {
    int data;
    struct list_node *tail;
};
typedef struct list_node list;

list *reverse(list *l) {
  list *r = l, *p = NULL;
  while (r != NULL) {
    list *q = r;
    r = r->tail;
    q->tail = p;
    p = q;
  }
  return p;
}
```

- **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)

Overview

Karlsruhe Institute of Technology

Thursday, August 30:

Part 4: Bounded model checking

Part 5: Improving scalability with light-weight program analysis methods

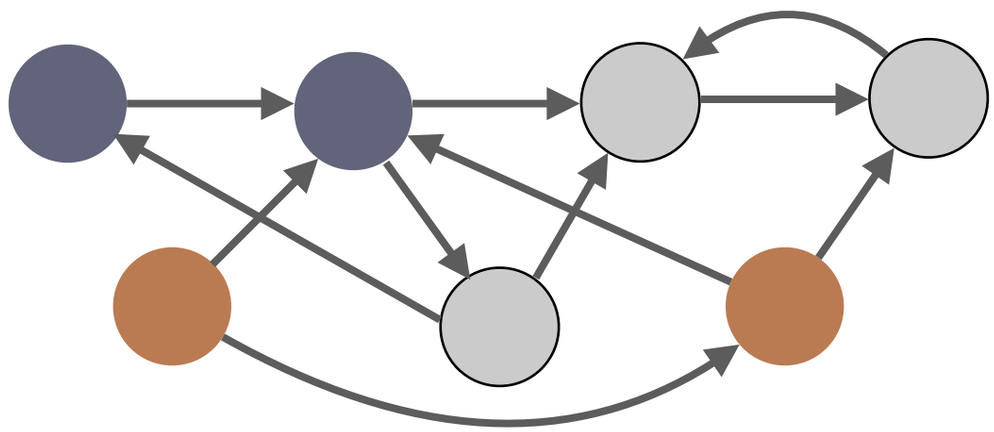
Part 6: Practical LLBMC

Working in groups on exercises

Part 4:
Bounded Model Checking

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:

	Hardware	Software
Elements	Flip-flops, registers, ...	Registers, memory, heap allocation state ...
State space	Cartesian product of Boolean variables	Cartesian product of integer variables of varying width
Transitions	Updates to registers / flip-flops	Updates of variables / memory

- 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 $I(s) \wedge T^*(s, s') \wedge B(s')$ 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

$$\text{BMC}_k: \quad I(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \wedge \bigwedge_{i=1}^{k} B(s_i)$$

  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 now 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'):

$$c_{\text{exec}}(I) \Rightarrow \neg \text{Err}(I) \wedge \text{Enc}(I)$$
$$c_{\text{exec}}(I') = c_{\text{exec}}(I) \wedge c_{\text{branch}}(I, I')$$

  - Here:
    - cexec is the execution condition of instruction I
    - cbranch is the condition when control flow goes from I to I'
    - Enc(I) is the encoding of the effects of I, 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) /\ x>5 /\ y<10 => 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 @iif : $(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
}
```

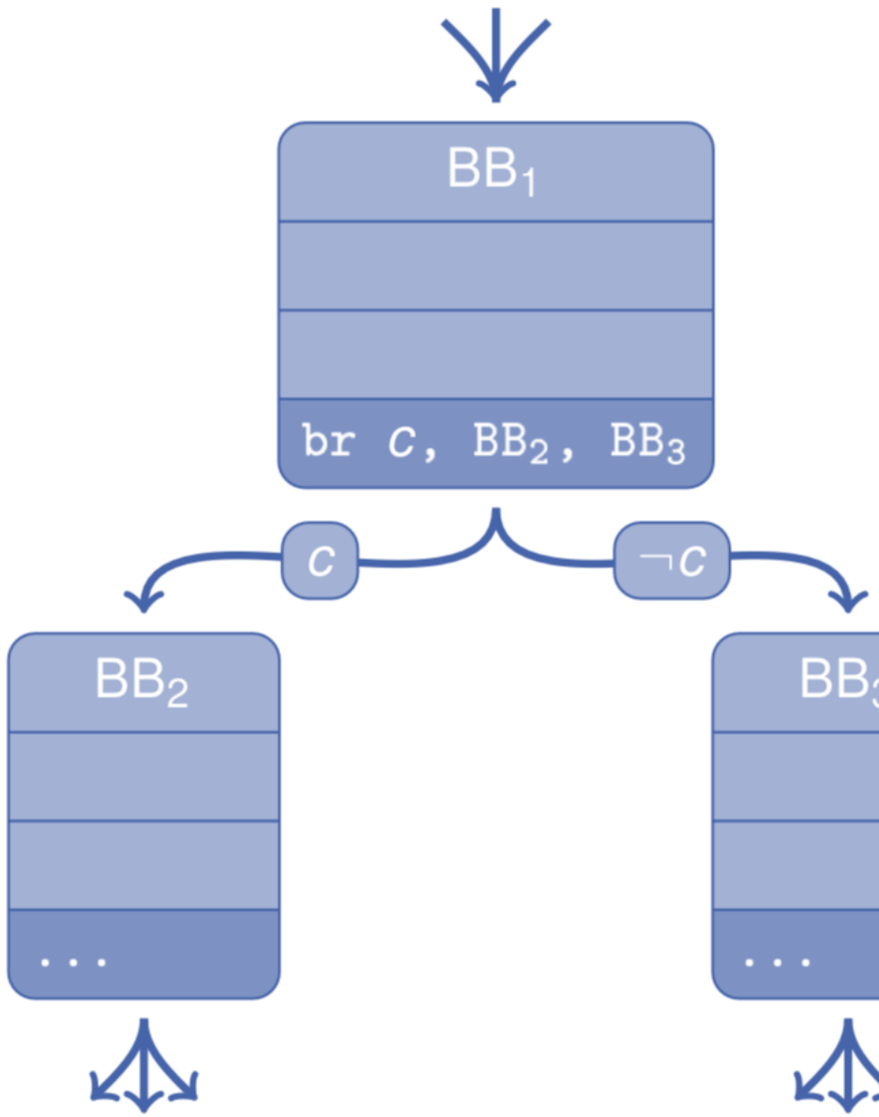
LLBMC Encoding: Basic Blocks

$$c_{exec}(entry) = true$$

LLBMC Encoding: Basic Blocks

$$c_{exec}(BB_2) = c_{exec}(BB_1)$$

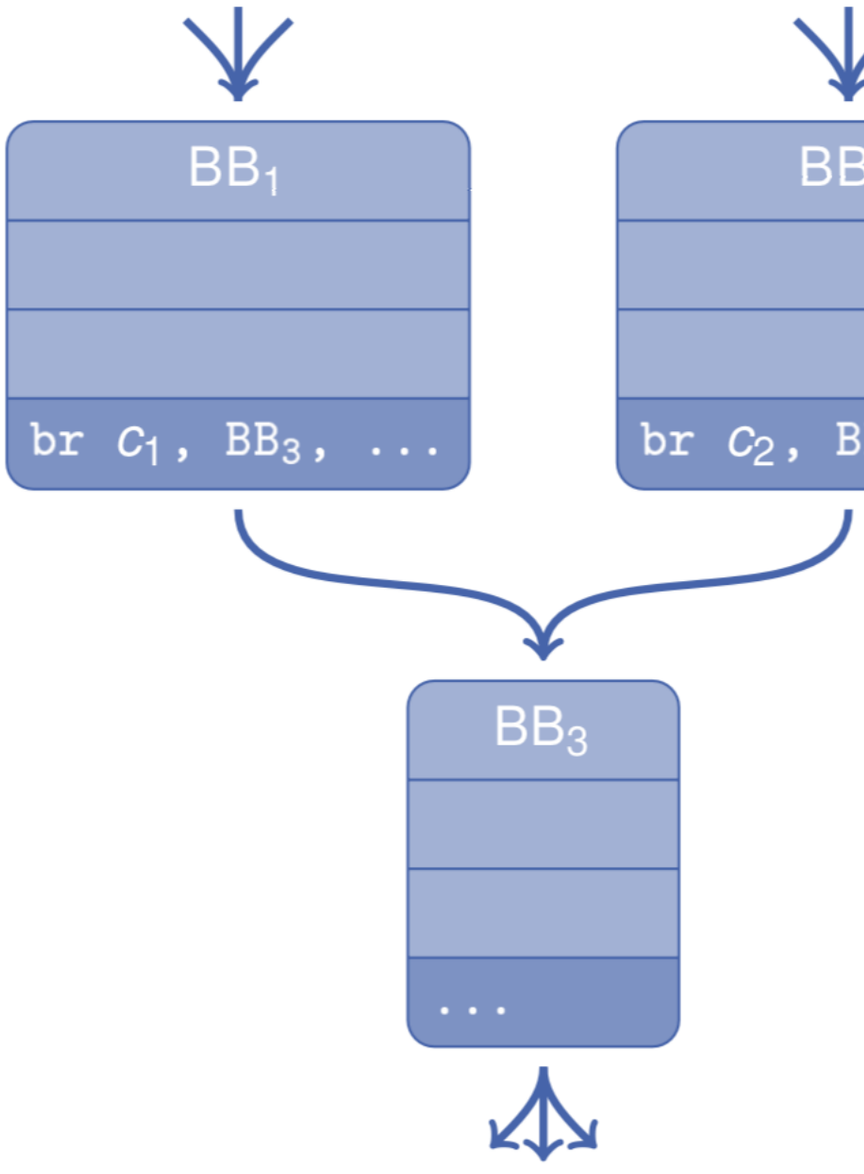
LLBMC Encoding: Basic Blocks



$$c_{exec}(BB_2) = c_{exec}(BB_1) \wedge c$$

$$c_{exec}(BB_3) = c_{exec}(BB_1) \wedge \neg c$$

LLBMC Encoding: Basic Blocks



$$c_{exec}(BB_3) = c_{exec}(BB_1) \wedge c_1 \vee c_{exec}(BB_2) \wedge c_2$$

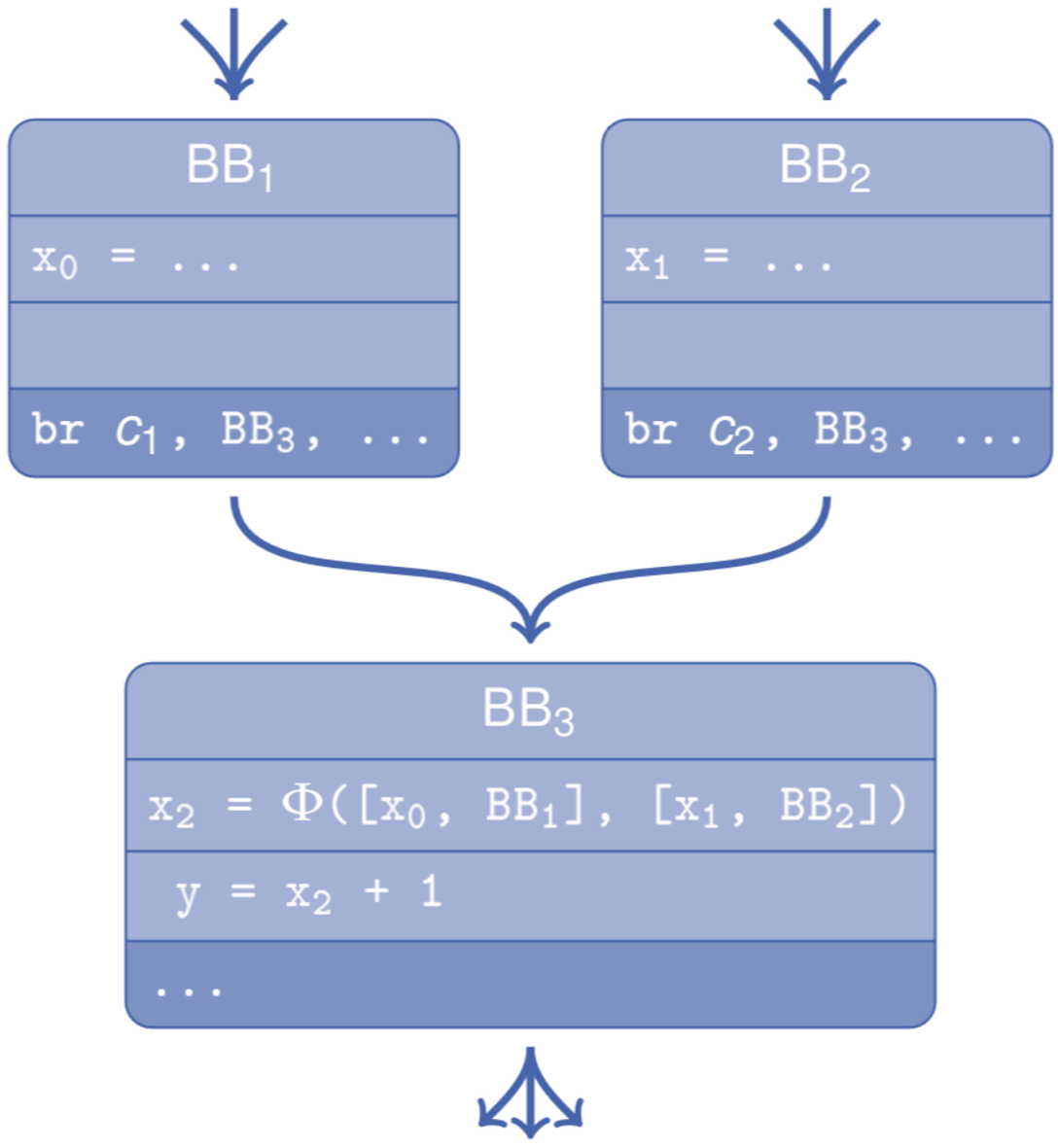
LLBMC Encoding: Instructions

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

```
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
}
```

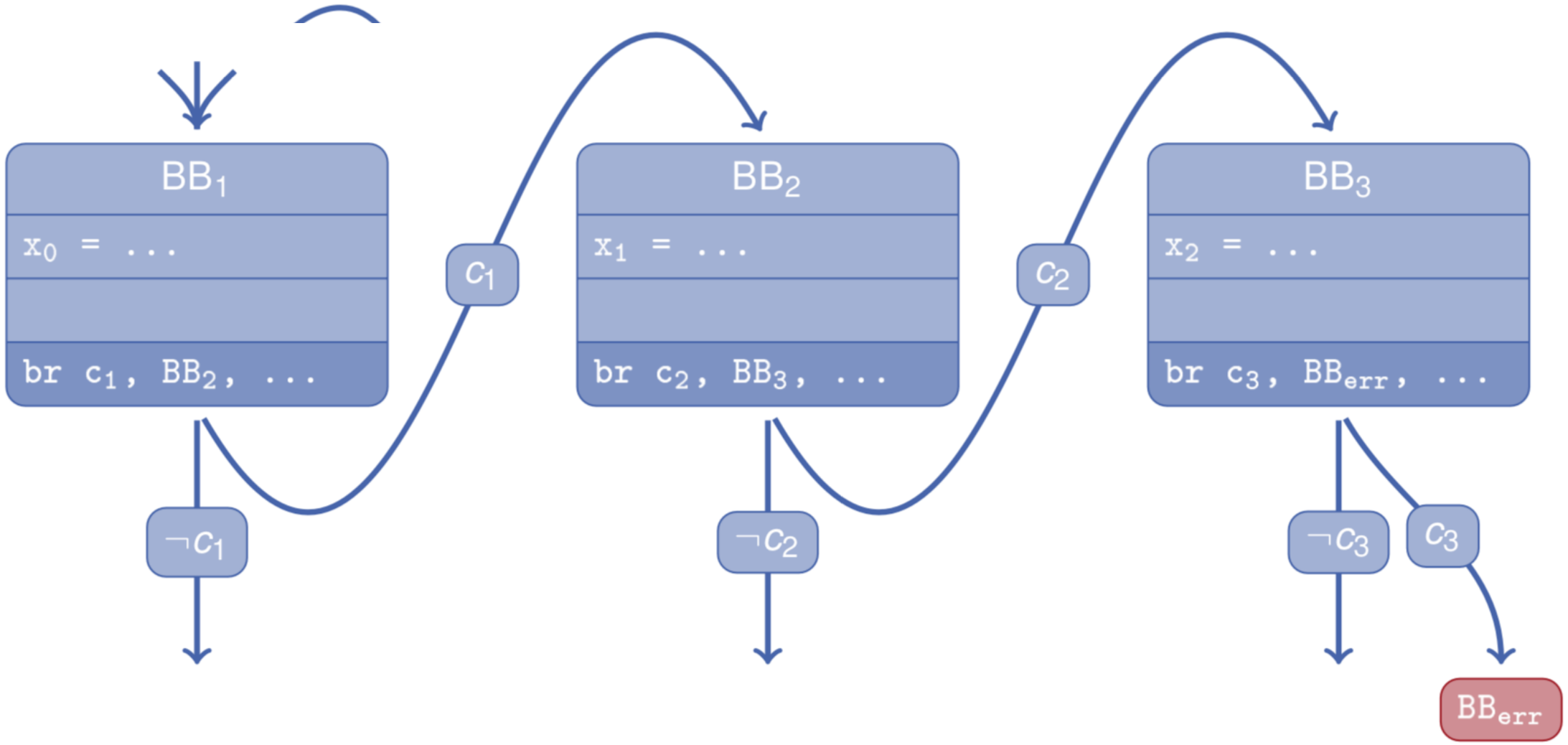
$$
\begin{aligned}
& sub = \texttt{bvsub}\ x\ y \\
\wedge\ & cmp = (\texttt{bvsgt}\ sub\ \texttt{bv}_{32,0})\ ?\ \texttt{bv}_{1,1}\ :\ \texttt{bv}_{1,0} \\
\wedge\ & conv = \texttt{zero\_extend}_{31}\ cmp \\
\wedge\ & cmp1 = (\texttt{bvsgt}\ x\ y)\ ?\ \texttt{bv}_{1,1}\ :\ \texttt{bv}_{1,0} \\
\wedge\ & conv2 = \texttt{zero\_extend}_{31}\ cmp1 \\
\wedge\ & cmp3 = (conv = conv2)\ ?\ \texttt{bv}_{1,1}\ :\ \texttt{bv}_{1,0} \\
\wedge\ & conv4 = \texttt{zero\_extend}_{31}\ cmp3
\end{aligned}
$$

LLBMC Encoding: Phi Nodes

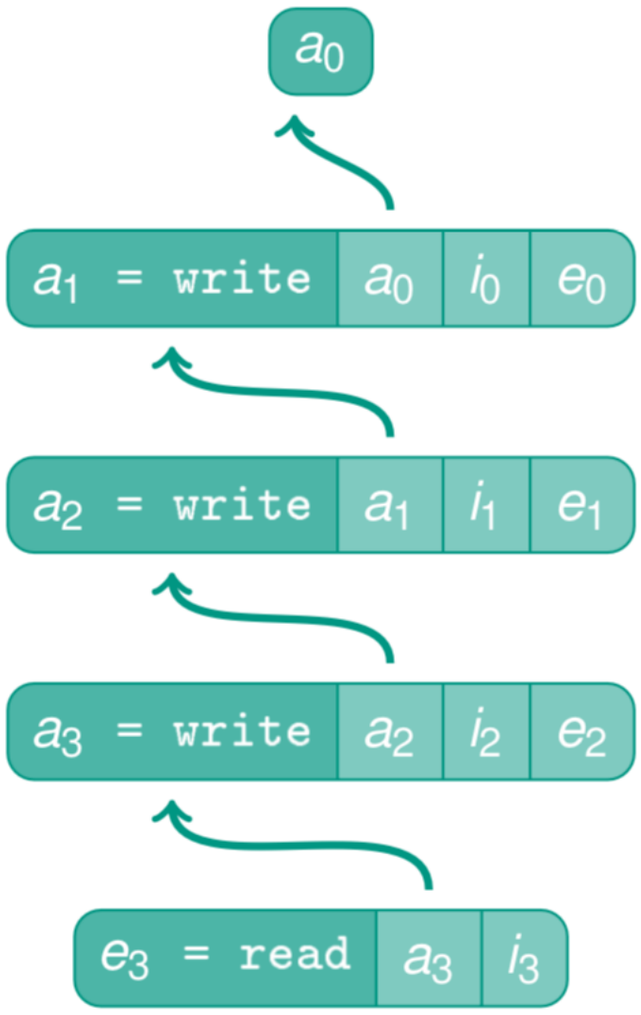


$$x_2 = c_{exec}(BB_1) \; ? \; x_0 \; : \; x_1$$

LLBMC Encoding: Loops



LLBMC Encoding: Memory Accesses

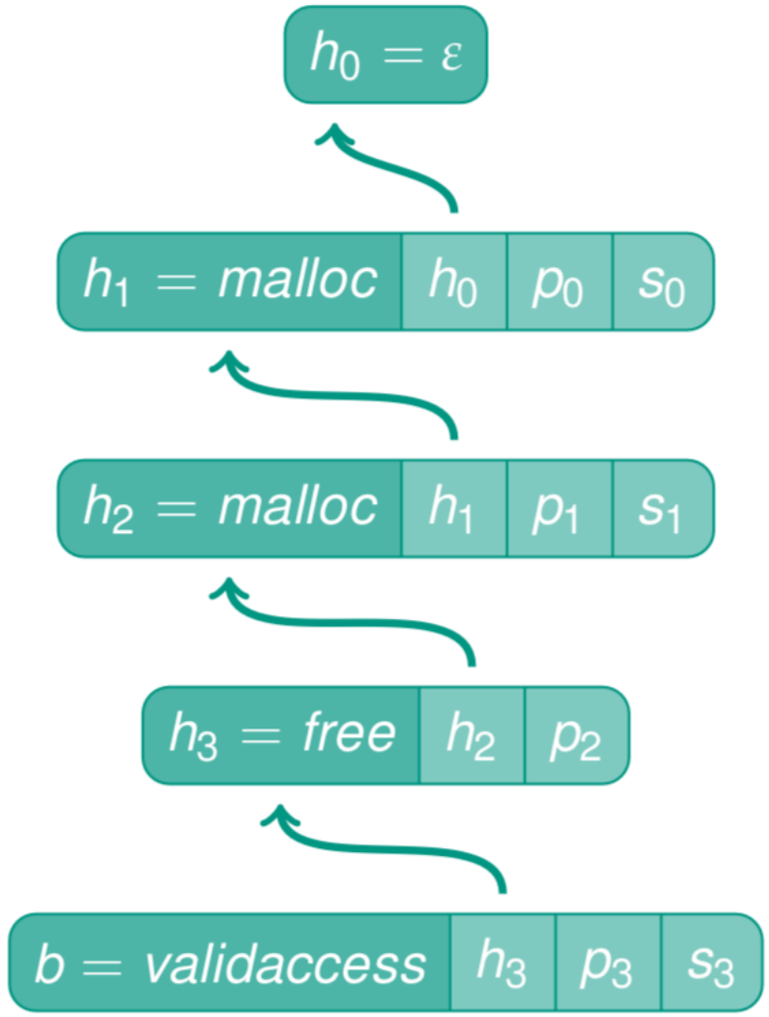


$$write : A \times I \times E \rightarrow A$$

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

$$read(write(write(write(a_0, i_0, e_0), i_1, e_1), i_2, e_2), i_3)$$

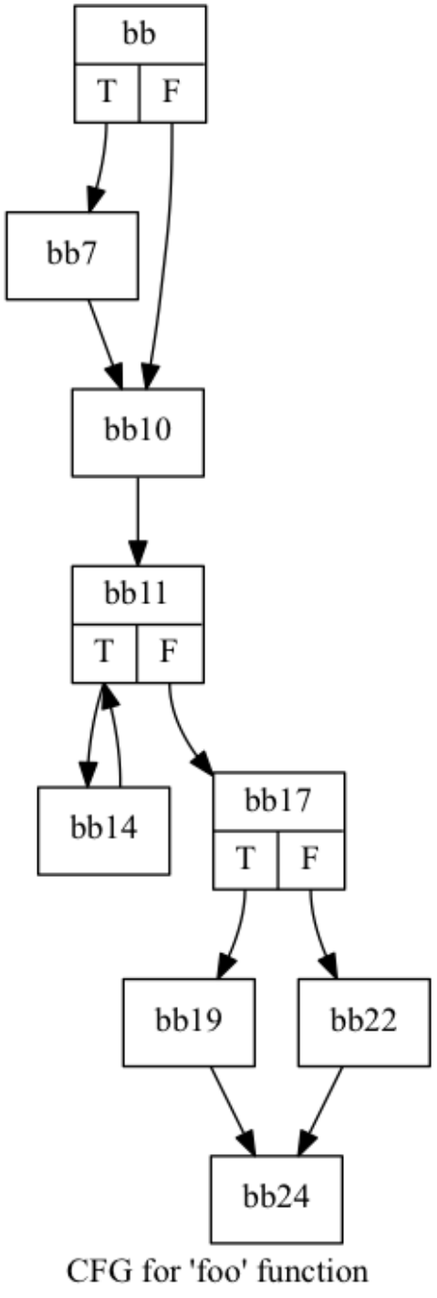
LLBMC Encoding: Heap State



Part 5:
Improving Scalability with Light-Weight Program-Analysis Methods

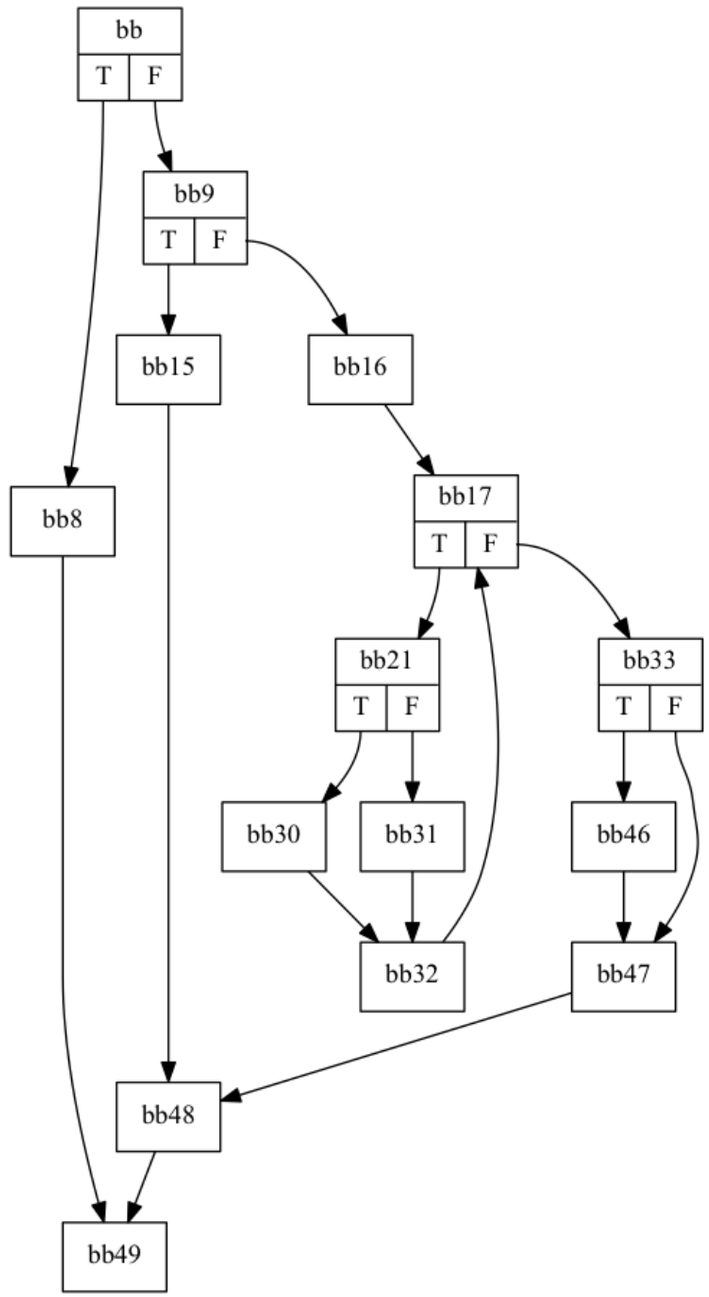
Backwards Slicing

```
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;
}
```



CFG for 'foo' function

Control Dependence Graph (CDG)



CFG for 'BINARYSEARCH_S16_Near_iL' function

Part 4:
Practical LLBMC

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                   - Mallocs 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
```