

The Bounded Model Checker LLBMC

- **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`)

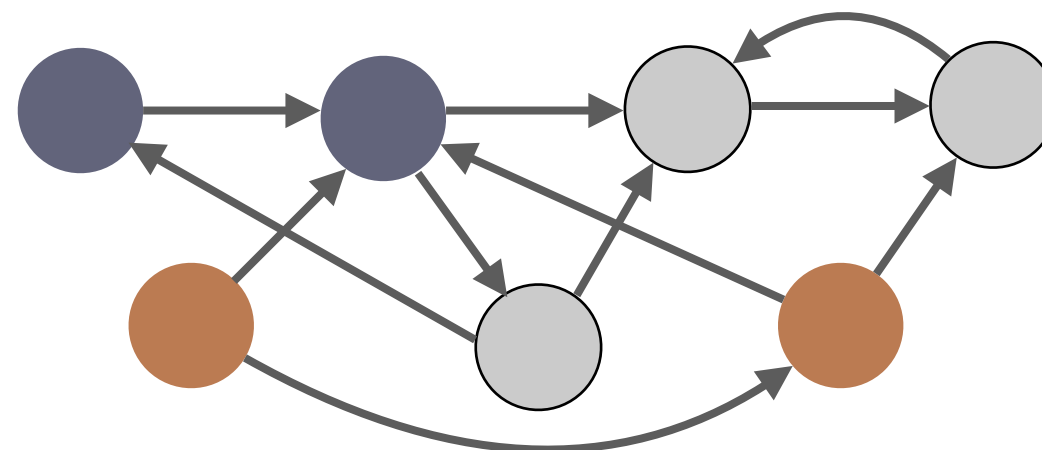
- **Employed techniques**

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

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 $(s_1, s_2, \dots, s_n, \dots)$ 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**

- **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:
 - $I(s) = (\neg b \wedge c)$, $B(s) = (b \wedge c)$
 - $T(s,s') = (b' \Leftrightarrow c) \wedge (c' \Leftrightarrow \neg(b \vee c))$

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:
  - $I(s) = (x=0 \wedge PC=0)$
  - $B(S) = (x>5)$
  - $T(s,s') = (PC=0 \Rightarrow x'=0 \wedge PC'=1) \wedge$   
 $(PC=1 \wedge x<4 \Rightarrow PC'=2 \wedge x'=x) \wedge$   
 $(PC=1 \wedge x \geq 4 \Rightarrow PC'=4 \wedge x'=x) \wedge$   
 $(PC=2 \Rightarrow \wedge PC'=3 \wedge x'=x+1) \wedge \dots$

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

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  $\text{BMC}_k$  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?)

- 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:
  - $c_{\text{exec}}$  is the execution condition of instruction  $I$
  - $c_{\text{branch}}$  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  $Li(x, \dots)$  for program locations.
- Then write each program transition as a rule like, e.g.,
  - $L1(x, y) \wedge x > 5 \wedge y < 10 \Rightarrow L2(x+1, y)$
- 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 @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
}
```

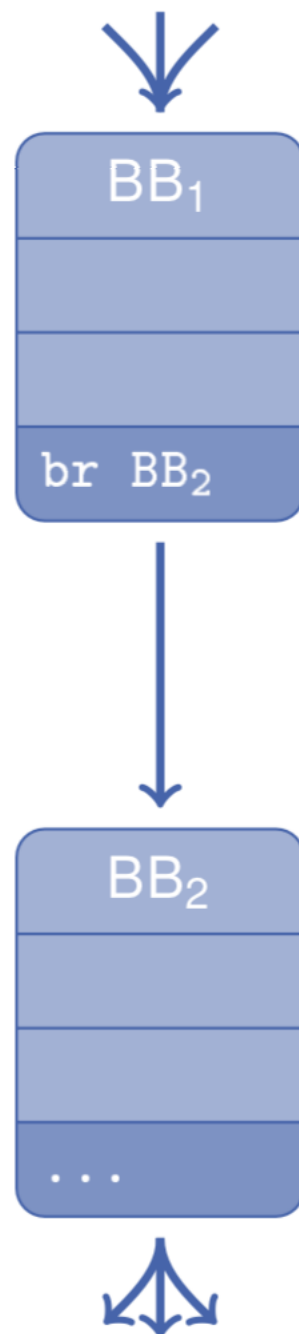


# LLBMC Encoding: Basic Blocks



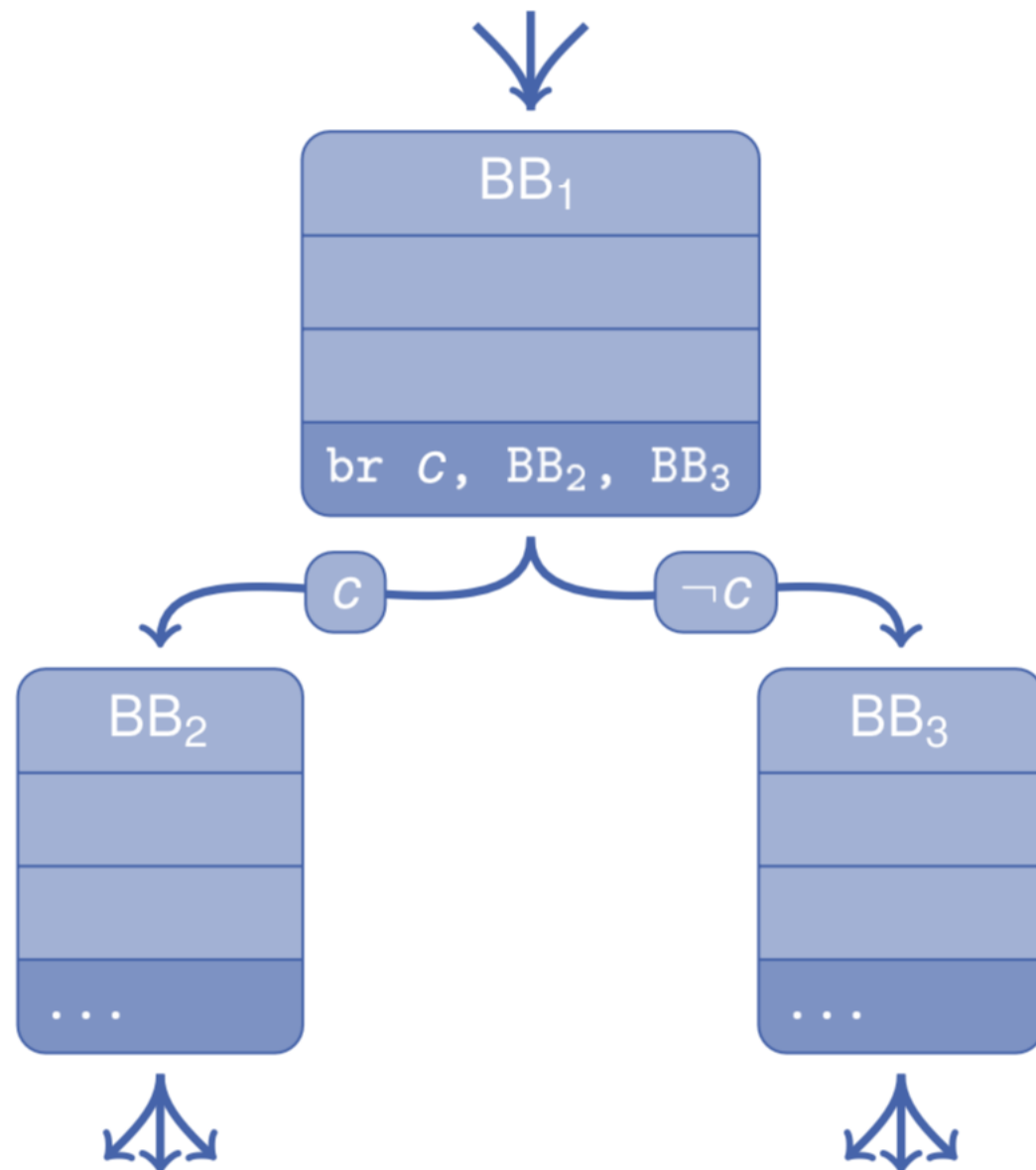
$$c_{exec}(entry) = true$$

# LLBMC Encoding: Basic Blocks



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

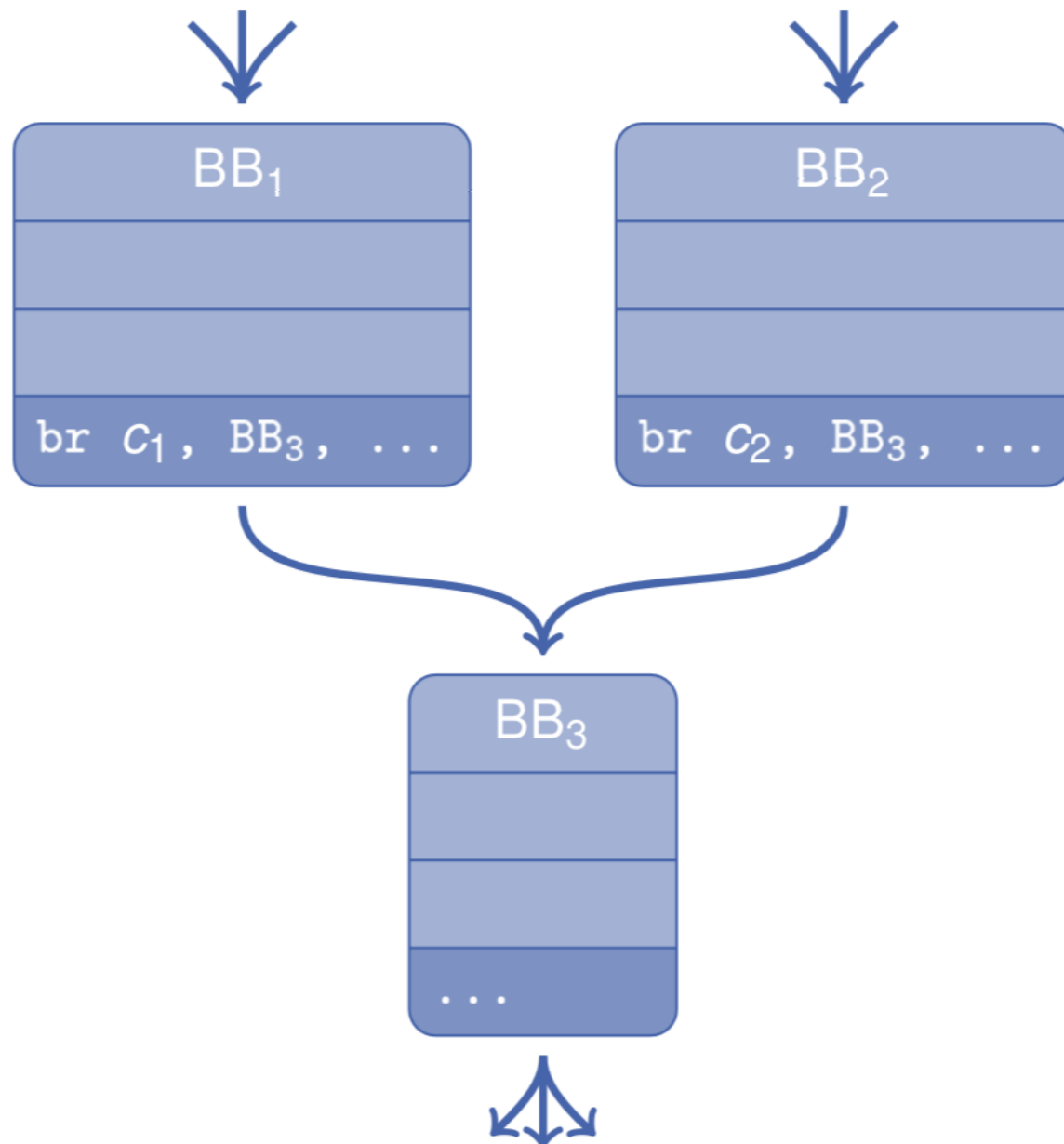
# LLBMC Encoding: Basic Blocks



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

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

# LLBMC Encoding: Basic Blocks



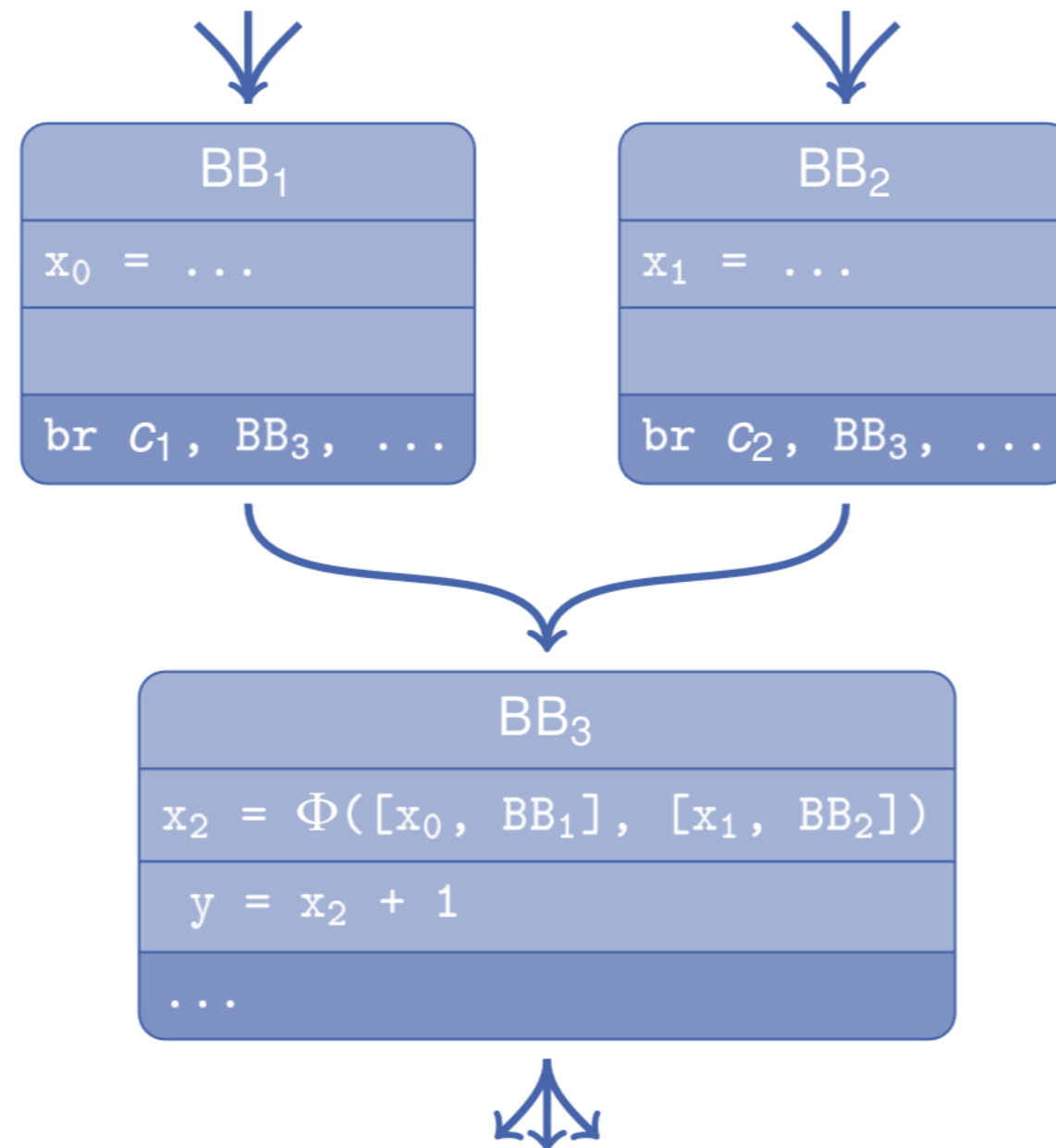
$$c_{exec}(BB_3) = c_{exec}(BB_1) \wedge c_1 \vee c_{exec}(BB_2) \wedge c_2$$

```
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
}
```

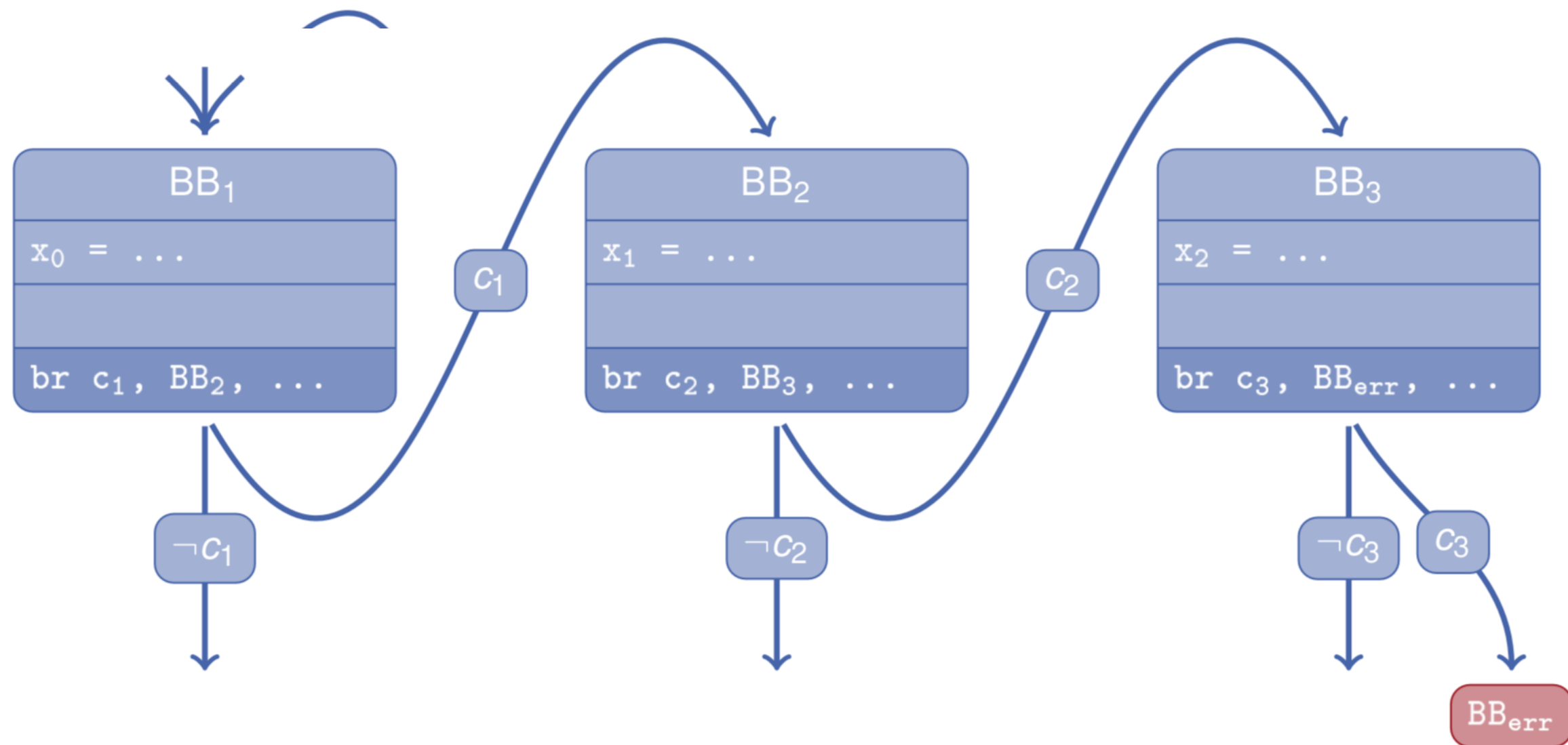
```
 sub = bvsb x y
^ cmp = (bvsgt sub bv32,0) ? bv1,1 : bv1,0
^ conv = zero_extend31 cmp
^ cmp1 = (bvsgt x y) ? bv1,1 : bv1,0
^ conv2 = zero_extend31 cmp1
^ cmp3 = (conv = conv2) ? bv1,1 : bv1,0
^ conv4 = zero_extend31 cmp3
```

# LLBMC Encoding: Phi Nodes



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

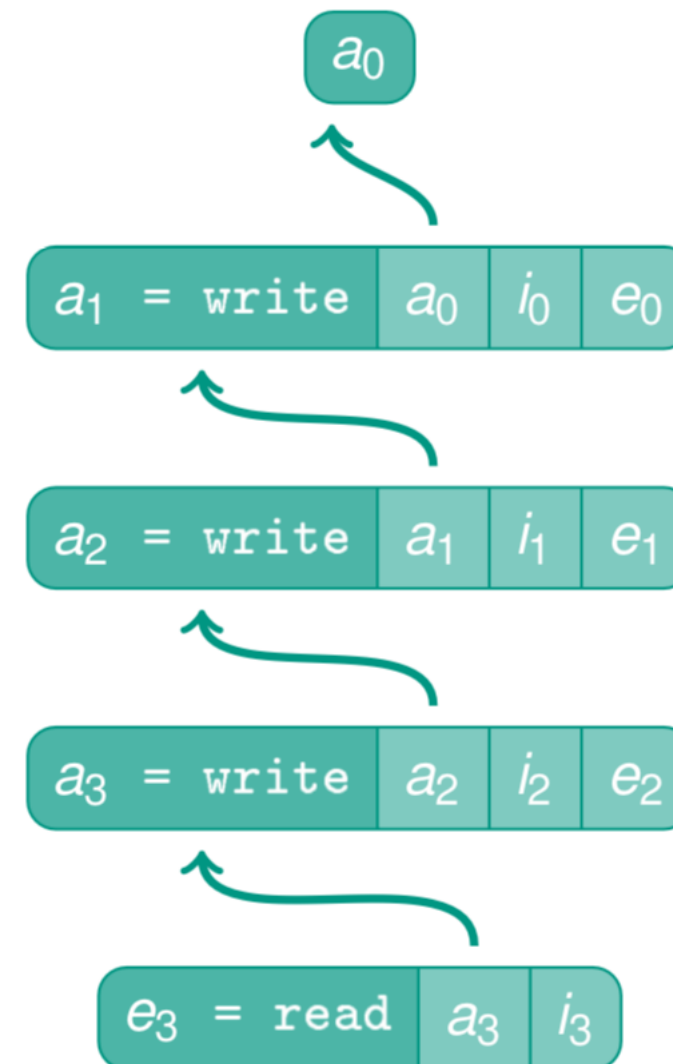
# LLBMC Encoding: Loops



# LLBMC Encoding: Memory Accesses

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

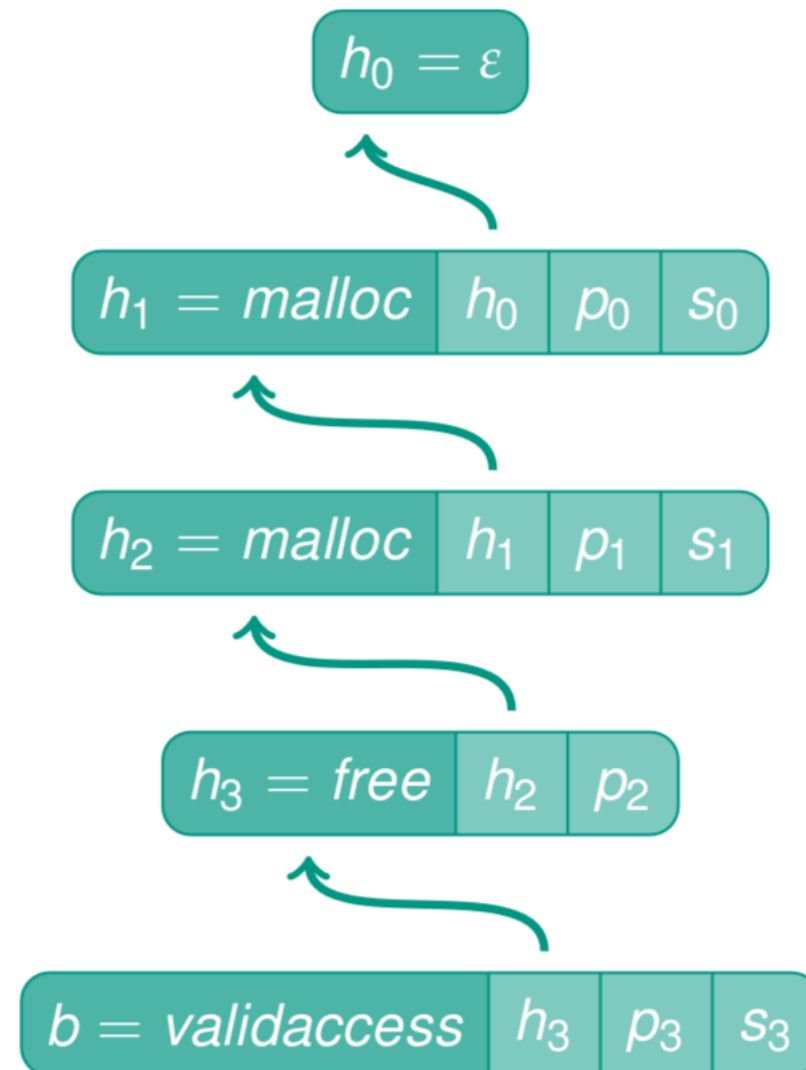
*read* :  $A \times I \rightarrow E$



$\text{read}(\text{write}(\text{write}(\text{write}(a_0, i_0, e_0), i_1, e_1), i_2, e_2), i_3)$



# LLBMC Encoding: Heap State



# Backwards Slicing

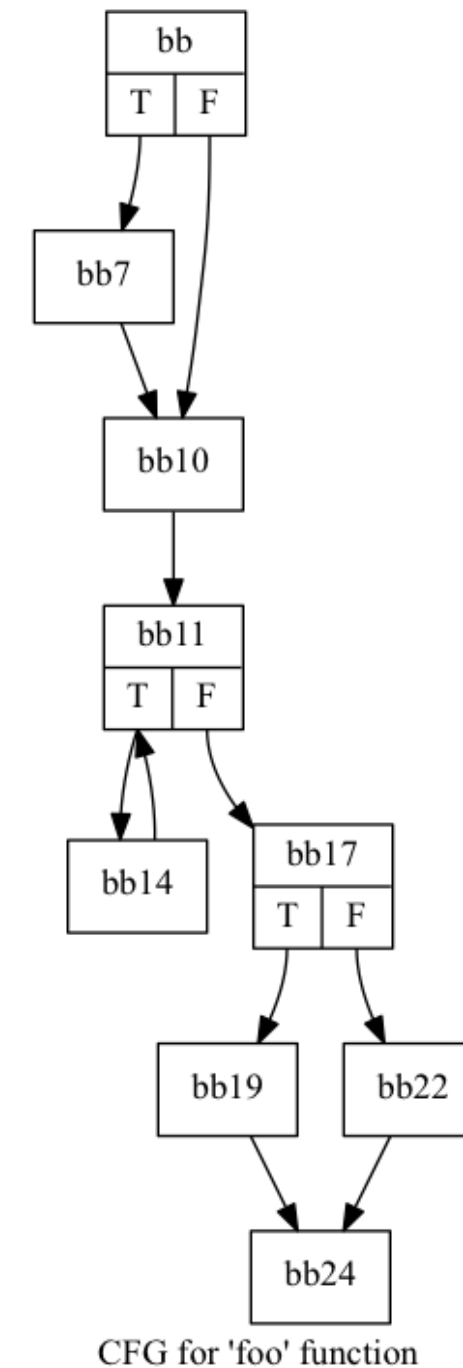
```
int a, b;

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

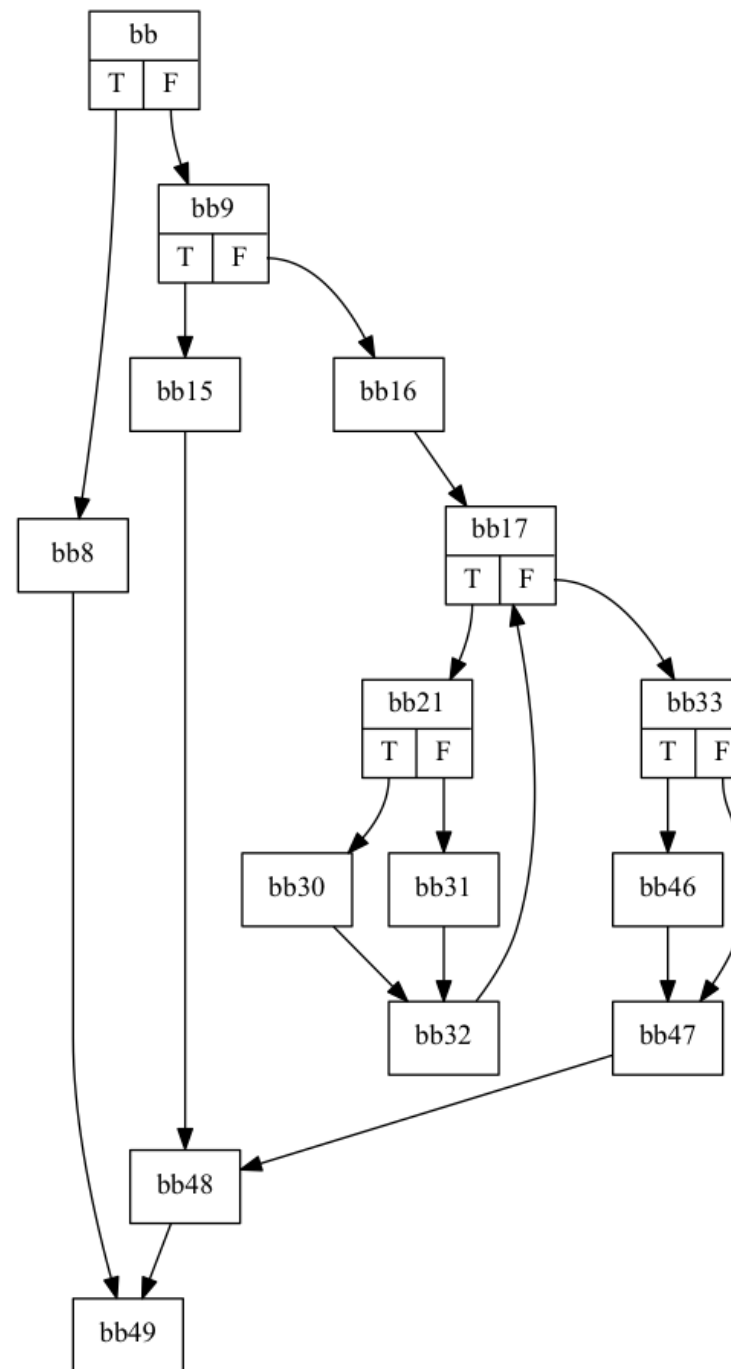
 while (t > a) {
 t -= 2;
 y++;
 }

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

 return x+b;
}
```



# 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:

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

# LLBMC Command Line Options

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

## 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                                  |