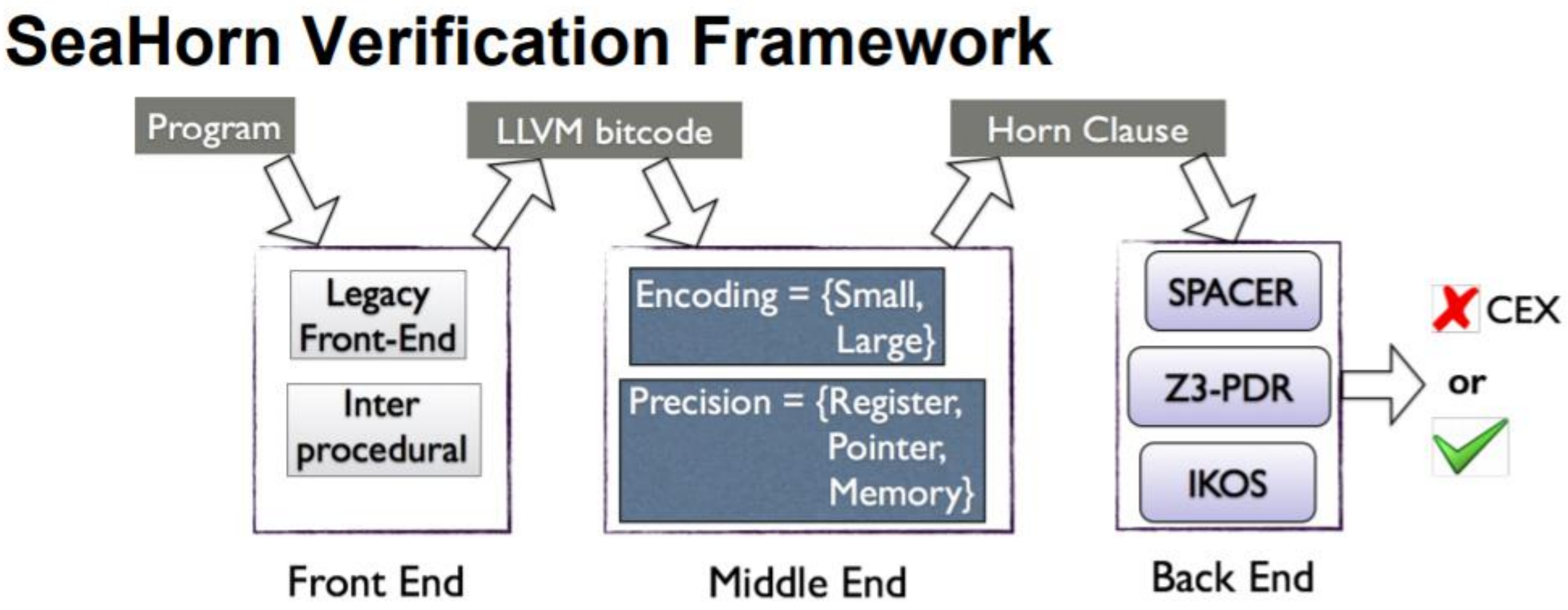
SeaHorn
A fully automated analysis framework for LLVM-based languages

Weizhi Feng

January 12, 2021

Overview

- SeaHorn is an automated analysis framework for LLVM-based languages.

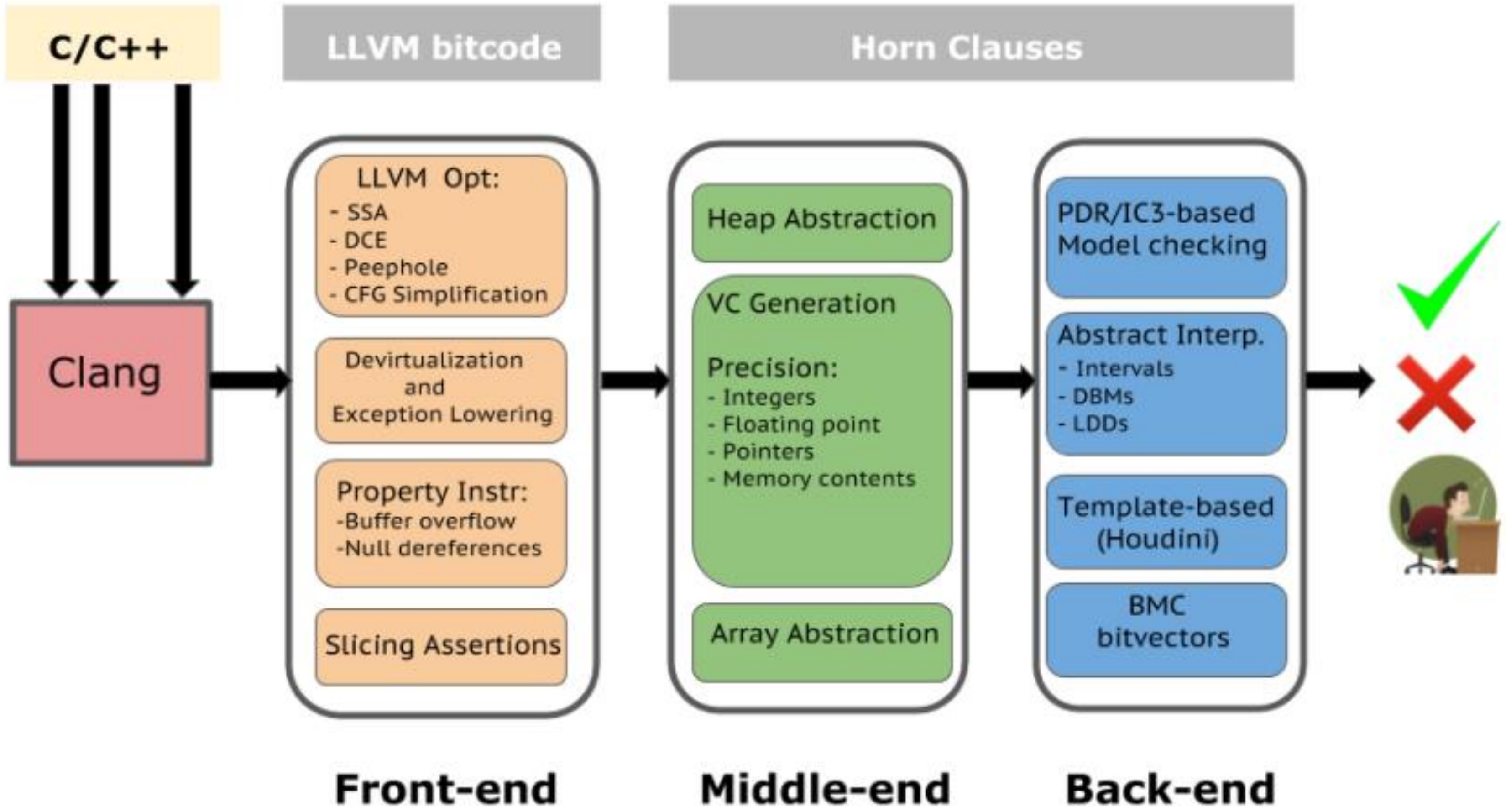


Overview

- SeaHorn is an automated analysis framework for LLVM-based languages.
- Distinguishing Features
  - LLVM front-end
  - Constrained horn clauses to represent verification conditons
  - Comparable to state-of-the-art tools at SV-COMP 15
- Usage
  - > sea pf FILE.c
    - ✓ Outputs sat for unsafe (has counterexample); unsat for safe.
  - Additional options
    - ✓ --cex=trace.xml outputs a counter-example in SV-COMP 15 format.
    - ✓ --track={reg,ptr,mem} track registers, points, memory content.
    - ✓ --step={large, small} verification condition step-semantics,
    - ✓ -small == basic block, large == loop-free control flow block.
    - ✓ --inline inline all functions in the front-end passes.

Workflow

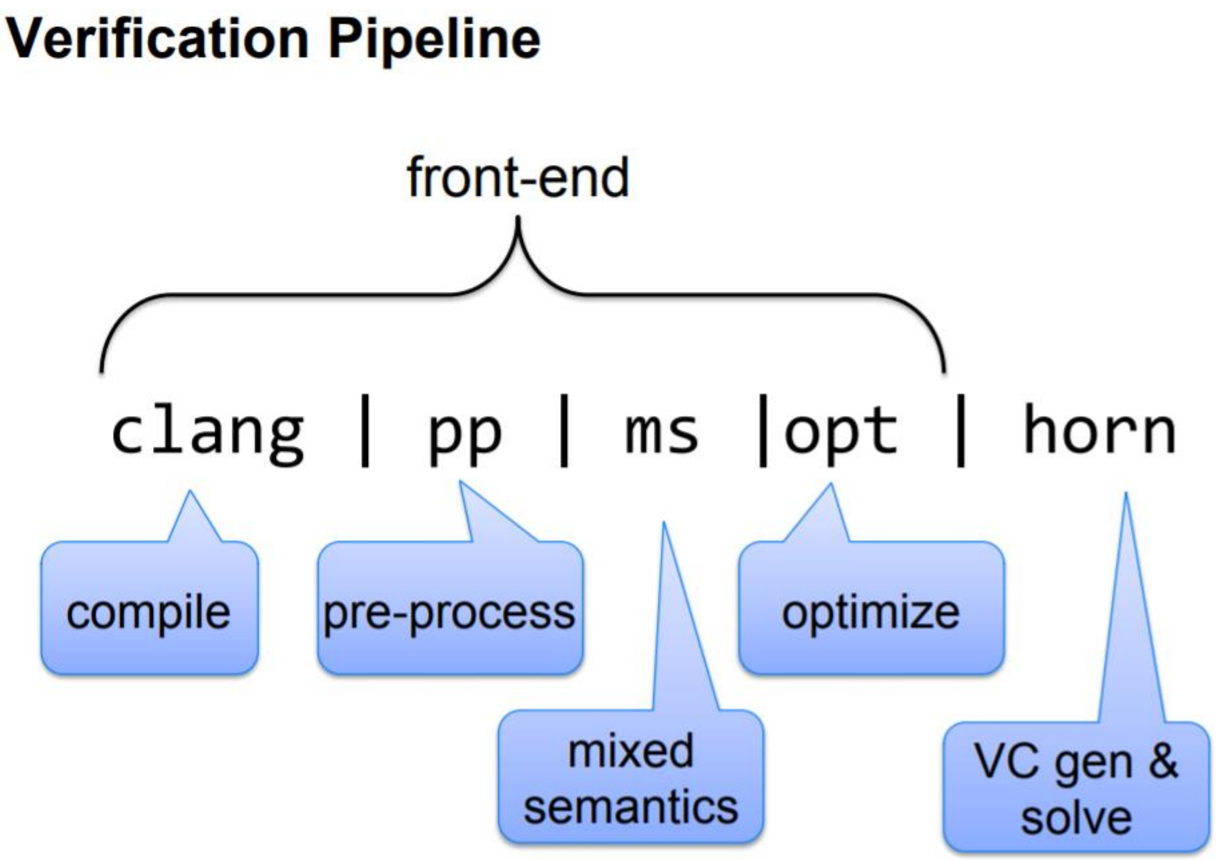
- Front-end
- Middle-end
- Back-end



Front-end

- Takes an LLVM based program (e.g., C) input program and generated LLVM IR bitcode.

- Specifically, it performs the pre-processing and optimization of the bitcode for verification purposes.



Pre-processing for Verification

- First, the input C program is pre-processed with CIL* to insert line markings for printing user-friendly counterexamples, define missing functions that are implicitly defined, and initialized all local variables.

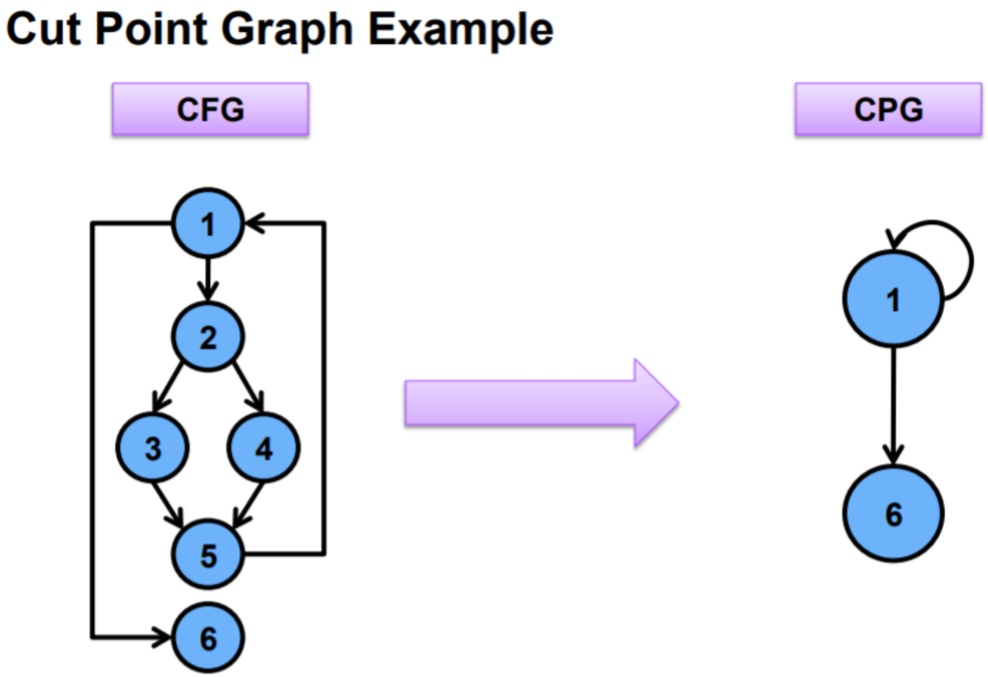
- Second, the result is translated into LLVM-IR bitcode, using llvm-gcc. After that, it performs compiler optimizations and preprocessing to simplify the verification task.
  - Function inlining, conversion to static single assignment (SSA) form, dead code elimination, peephole optimizations, CFG simplifications, etc.

- Make SeaHorn not to be limited to C programs, but applicable to a broader set of languages based on LLVM (C++, Objective C and Swift).

* intermediate language and tools for analysis and transformation of C programs

From CFG to Cut Point Graph

- A Cut Point Graph hides (summarizes) fragments of a control flow graph by (summary) edges.
- Vertices correspond to some basic blocks.
- An edge between cut-points c and d summarizes all finite (loop-free) executions from c to d that do not pass through any other cut-points.



Mixed semantics

- One typical problem in proving safety of large programs is that assertions can be nested very deep inside the call graph. As a result, counterexamples are longer and it is harder to decide for the verification engine what is relevant for the property of interest.
- To mitigate this problem, the front-end provides a transformation based on the concept of mixed semantics.

- \- if P may fail, then make a copy of P's body (in main) and jump to the copy.
- \- if P may succeed, then make the call to P as usual. Since P is known not to fail each assertion in P can be safety replaced with an assume.

- Upon completion, only the main function has assertions and each procedure is inlined at most once.

Mixed semantics - example

- A main procedure calling two other procedures p_1 and p_2 with three assertions c_1, c_2, and c_3.
- The new program after the mixed-semantics transformation.

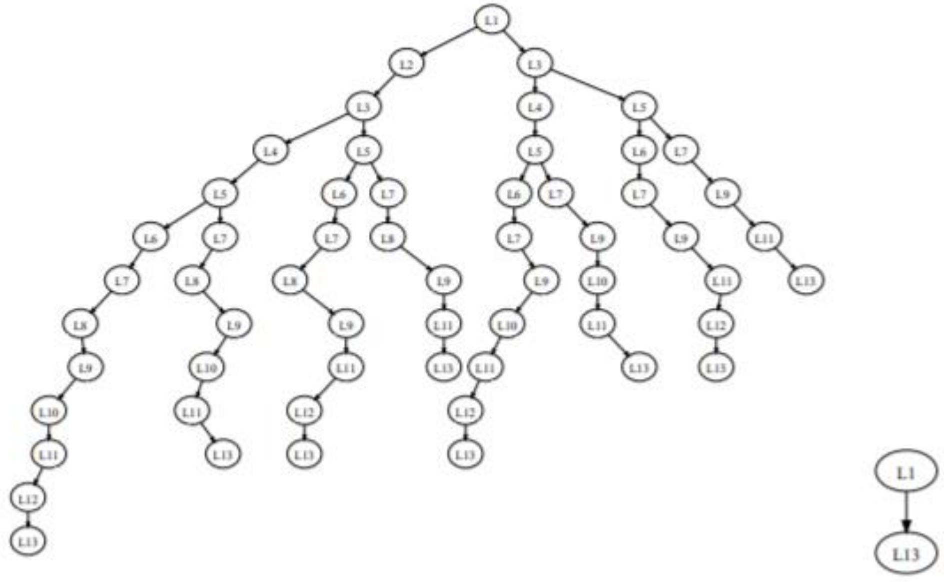
$main$ ()	$main_{new}$ ()	$p1_{entry}$:	$p1_{new}$ ()
$p1$ (); $p1$ ();	if (*) goto $p1_{entry}$;	if (*) goto $p2_{entry}$;	$p2_{new}$ ();
assert ($c1$);	else $p1_{new}$ ();	else $p2_{new}$ ();	assume ($c2$);
$p1$ ()	if (*) goto $p1_{entry}$;	if ($\neg c2$) goto $error$;	$p2_{new}$ ()
$p2$ ();	else $p1_{new}$ ();	$p2_{entry}$:	assume ($c3$);
assert ($c2$);	if ($\neg c1$) goto $error$;	if ($\neg c3$) goto $error$;	
$p2$ ()	assume (false);	assume (false);	
assert ($c3$);		$error$: assert (false);	

Fig. 2: A program before and after mixed-semantics transformation.

Middle-end

- Encoding verification conditions:
  - SeaHorn provides two different semantics encodings: a) a small step encoding and b) a large-block encoding (LBE).
  - Different degree of precision: LBE is often more efficient but small-step might be more useful if a counterexample is needed.

```
L1:  if(p1) {
L2:    x1 = 1;
     }
L3:  if(p2) {
L4:    x2 = 2;
     }
L5:  if(p3) {
L6:    x3 = 3;
     }
L7:  if(p1) {
L8:    if (x1 != 1) goto ERR;
     }
L9:  if (p2) {
L10:   if (x2 != 2) goto ERR;
     }
L11: if (p3) {
L12:   if (x3 != 3) goto ERR;
     }
L13: return EXIT_SUCCESS;
ERR: return EXIT_FAILURE;
```



(a) Example C program (b) ART for SBE (c) ART for LBE

Constrained Horn Clauses (CHC)

- A Constrained Horn Clause (CHC) is a FOL formula of the form:

$$\forall V \,.\, (\phi \wedge p_1[X_1] \wedge \ldots \wedge p_n[X_n] \rightarrow h[X]),$$

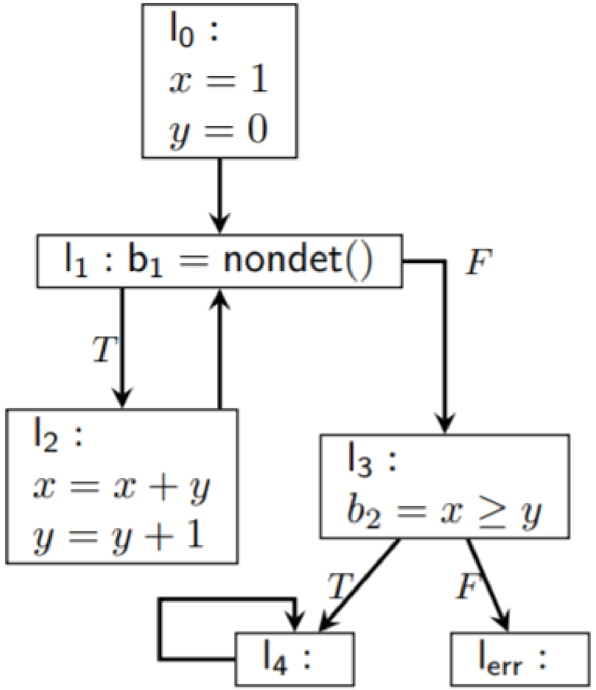
where

- A is a background theory (e.g., Linear Arithmetic, Arrays, Bit-Vectors, or combinations of the above)
- ϕ is a constrained in the background theory A
- $p_1, \ldots, p_n$, h are n-ary predicates
- $p_i[X]$ is an application of a predicate to first-order terms

Example – small-step encoding of VCs using Horn clauses

- Program -> Control-Flow -> Verification Conditions

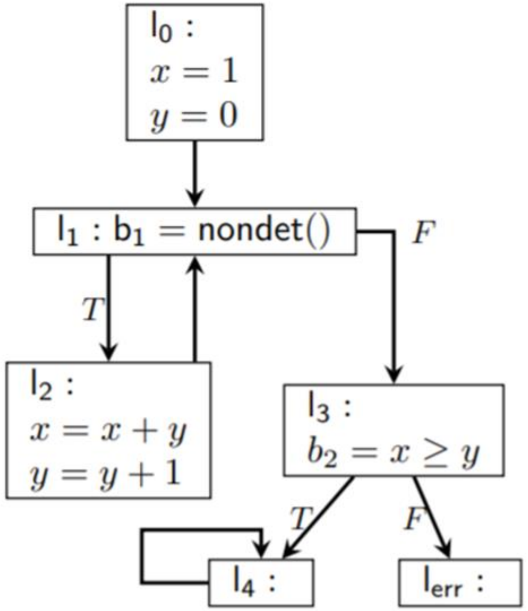
$\text{int } x = 1;$
$\text{int } y = 0;$
$\text{while } (*)\ \{$
$\quad x = x + y;$
$\quad y = y + 1;$
$\}$
$\text{assert}(x \geq y);$



$\langle 1 \rangle\ p_0.$
$\langle 2 \rangle\ p_1(x, y) \leftarrow$
$\qquad p_0, x = 1, y = 0.$
$\langle 3 \rangle\ p_2(x, y) \leftarrow p_1(x, y)\ .$
$\langle 4 \rangle\ p_3(x, y) \leftarrow p_1(x, y)\ .$
$\langle 5 \rangle\ p_1(x', y') \leftarrow$
$\qquad p_2(x, y),$
$\qquad x' = x + y,$
$\qquad y' = y + 1.$
$\langle 6 \rangle\ p_4 \leftarrow (x \geq y), p_3(x, y).$
$\langle 7 \rangle\ p_{err} \leftarrow (x < y), p_3(x, y).$
$\langle 8 \rangle\ p_4 \leftarrow p_4.$
$\langle 9 \rangle\ \bot \leftarrow p_{err}.$

Example – small-step encoding of VCs using Horn clauses

- The set of CHCs essentially represents the small-step operational semantics of CFG.
- Each basic block is encoded as a Horn clause.
- A basic block label l_i in the CFG is translated into $p_i(X_1, \dots, X_n)$.



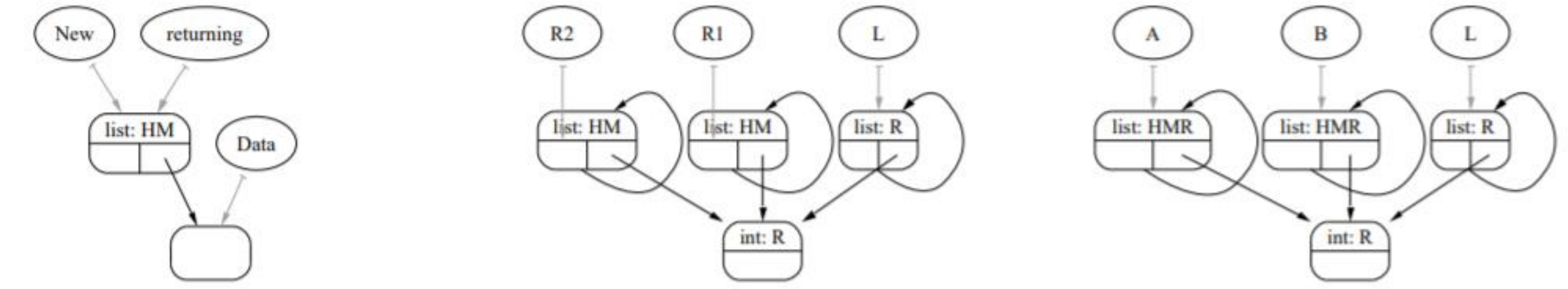
$\langle 1 \rangle$ p_0.
$\langle 2 \rangle$ $\mathsf{p}_1(x, y) \leftarrow$
$\quad \mathsf{p}_0, x = 1, y = 0.$
$\langle 3 \rangle$ $\mathsf{p}_2(x, y) \leftarrow \mathsf{p}_1(x, y)$.
$\langle 4 \rangle$ $\mathsf{p}_3(x, y) \leftarrow \mathsf{p}_1(x, y)$.
$\langle 5 \rangle$ $\mathsf{p}_1(x', y') \leftarrow$
$\quad \mathsf{p}_2(x, y),$
$\quad x' = x + y,$
$\quad y' = y + 1.$
$\langle 6 \rangle$ $\mathsf{p}_4 \leftarrow (x \geq y), \mathsf{p}_3(x, y).$
$\langle 7 \rangle$ $\mathsf{p}_{err} \leftarrow (x < y), \mathsf{p}_3(x, y).$
$\langle 8 \rangle$ $\mathsf{p}_4 \leftarrow \mathsf{p}_4.$
$\langle 9 \rangle$ $\perp \leftarrow \mathsf{p}_{err}.$

Middle-end

- SeaHorn middle-end offers a very simple interface for developers to implement an encoding of the verification semantics that fits their needs.
- Symbolic store:
  - the core of the SeaHorn middle-end.
  - A symbolic store simply maps program variables to symbolic values.
  - The small-step verification semantics is provided by implementing a symbolic execution interface.
- SeaHorn middle-end includes verification semantics with different levels of abstraction:
  - Registers only: only models LLVM numeric registers.
  - Registers + Pointers (without memory content): models numeric and pointer registers.
  - Registers + Pointers + Memory: models numeric and pointer registers and the heap. The heap is modeled by a collection of non-overlapping arrays.

Middle-end

- Data structure analysis (DSA): a heap analysis to model heap.
- The analysis build for each function a DS graph where each node represents a potentially infinite set of memory objects and distinct DSA nodes express disjoint sets of objects. Edges in the graph represents points-to relationships between DS nodes.
- Given a DS graph we can map each DS node to an array. Then each memory load (read) and store (write) in the LLVM bitcode can be associated with a particular DS node.



(a) DS Graph for createnode (b) DS Graph for splitclone (c) DS Graph for processlist

Verification Engines

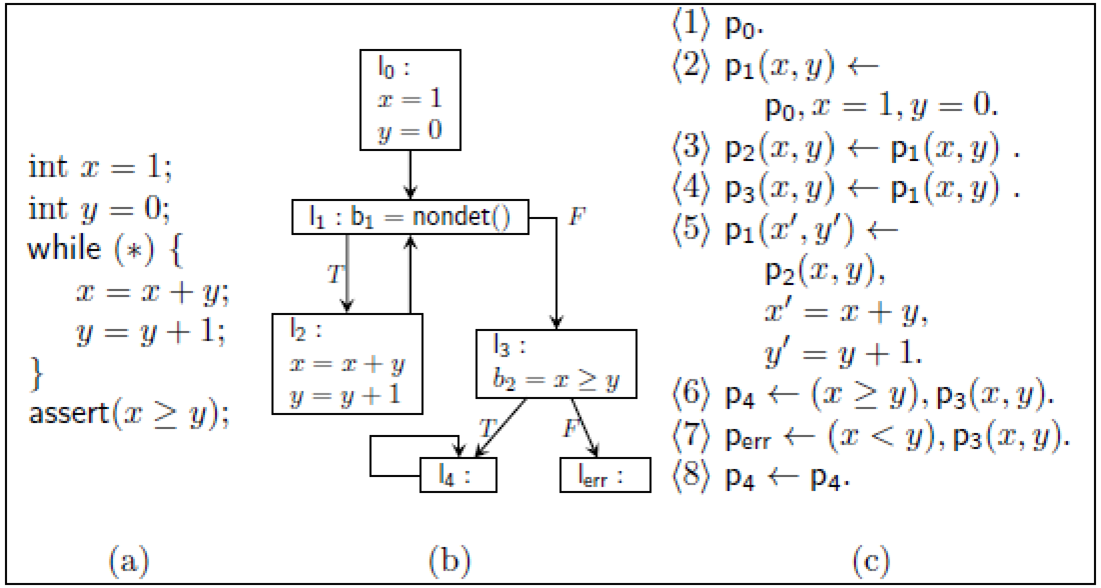
- Horn clause-based verification tool.
- SMT-Based Model Checking with SPACER
- Abstract Interpretation with IKOS

SPACER: Solving CHC in Z3

- Spacer: solver for SMT-constrained Horn Clauses
- Support for Non-linear CHC
- Support SMT-Theories

IKOS: an open-source library of abstract domains with a SOTA fixed-point algorithm

- SeaHorn users can choose IKOS as the only back-end engine to discharge proof obligations.
- In the example, SPACER alone can discover $x \geq y$ but it misses the vital invariant $y \geq 0$. Thus, it does not terminate. On the contrary, IKOS alone with the abstract domain of DBMs(Difference-Bound Matrix) can prove safety immediately.



Experimental Evaluation

- Results of SV-COMP 2015

SV-COMP 2015

http://sv-comp.sosy-lab.org/2015/

4th Competition on Software Verification held (here!) at TACAS 2015

Goals

- Provide a snapshot of the state-of-the-art in software verification to the community.
- Increase the visibility and credits that tool developers receive.
- Establish a set of benchmarks for software verification in the community.

Participants:

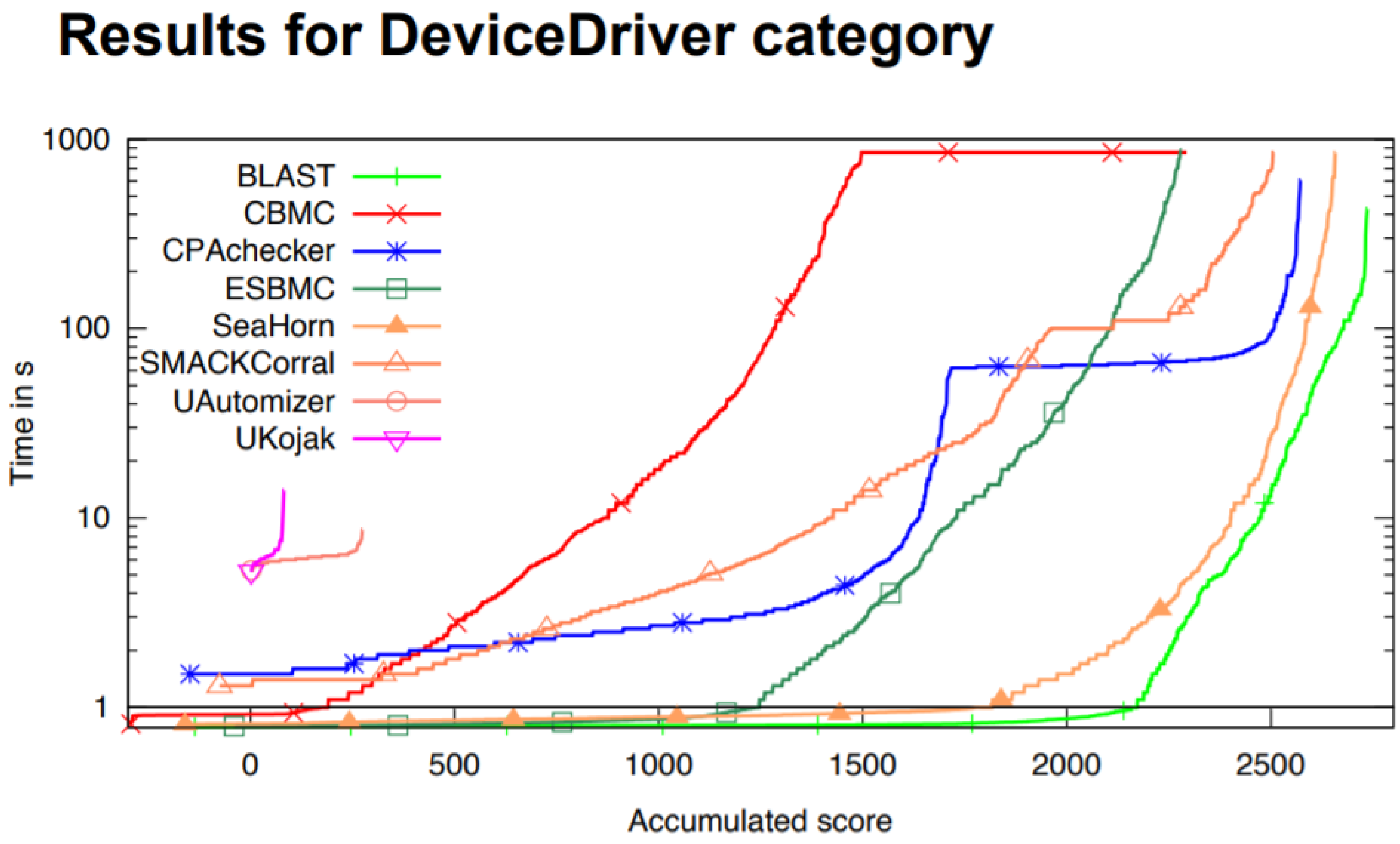
- Over 22 participants, including most popular Software Model Checkers and Bounded Model Checkers

Benchmarks:

- C programs with error location (programs include pointers, structures, etc.)
- Over 6,000 files, each 2K – 100K LOC
- Linux Device Drivers, Product Lines, Regressions/Tricky examples
- http://sv-comp.sosy-lab.org/2015/benchmarks.php

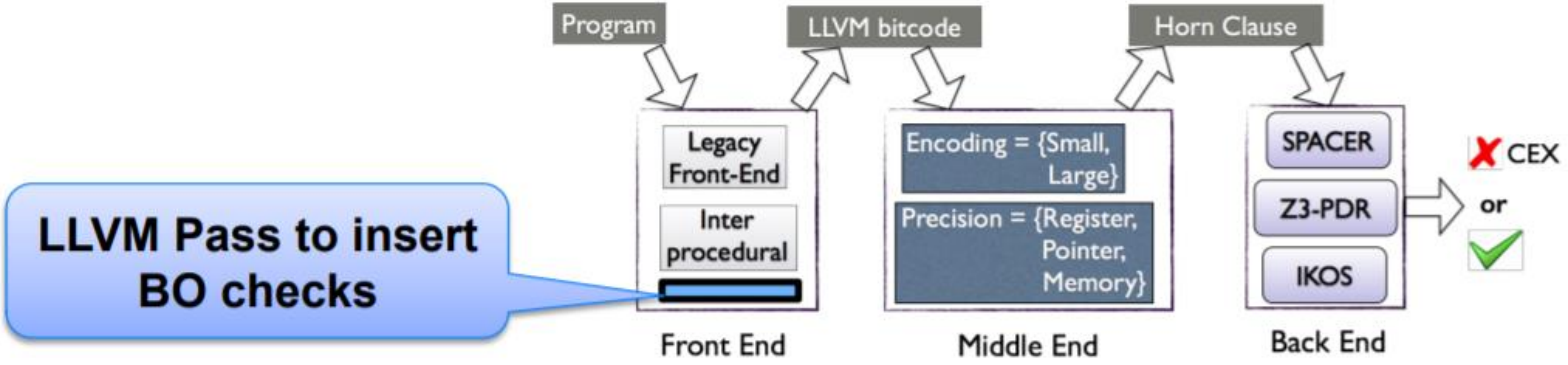
Experimental Evaluation

- Results of SV-COMP 2015



Case Study: Checking buffer overflow in Avionics Software

- Evaluated the SeaHorn built-in buffer overflow checks on two autopilot control software (paparazzi and mnav autopilots).
- To prove absence of buffer overflows, they only need to add in the front-end a new LLVM transformation pass that inserts the corresponding checks in the bitcode.



Case Study: Checking buffer overflow in Avionics Software

- For each pointer dereference *p, add two shadow registers: p.offset and p.size.
- For each *p, add two assertions:
  - assert $(\mathrm{p.\,offset} \geq 0)$ (underflow)
  - assert $(\mathrm{p.\,offset} < \mathrm{p.\,size})$ (overflow)