Connecting LLVM with a WCET tool

Rick Veens
rickveens92@gmail.com

FOSDEM 2018 (4-2-2018)
10:35 – 11:05

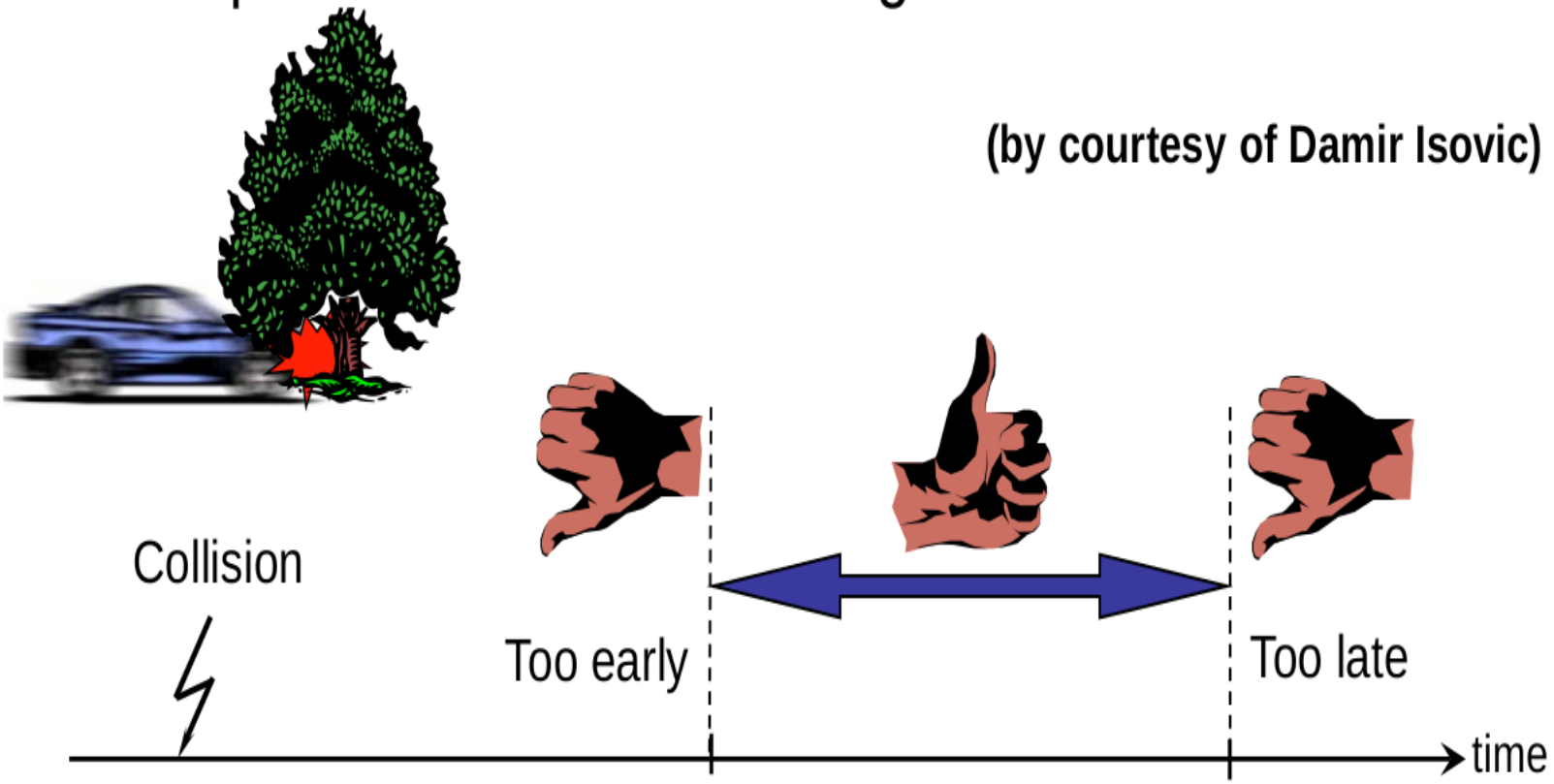
“Connecting LLVM with a WCET tool” talk outline

- What is WCET? WCET analysis?
- Why connect LLVM with a WCET tool?
- What tool to pick? (SWEET)
- Approach to connecting SWEET and LLVM
- WCET for the ARM Cortex-M3
- Conclusions

What is WCET?

What is WCET?

- Worst Case Execution Time: longest path in the code
  - Example: inflation of an air bag



How to determine WCET?

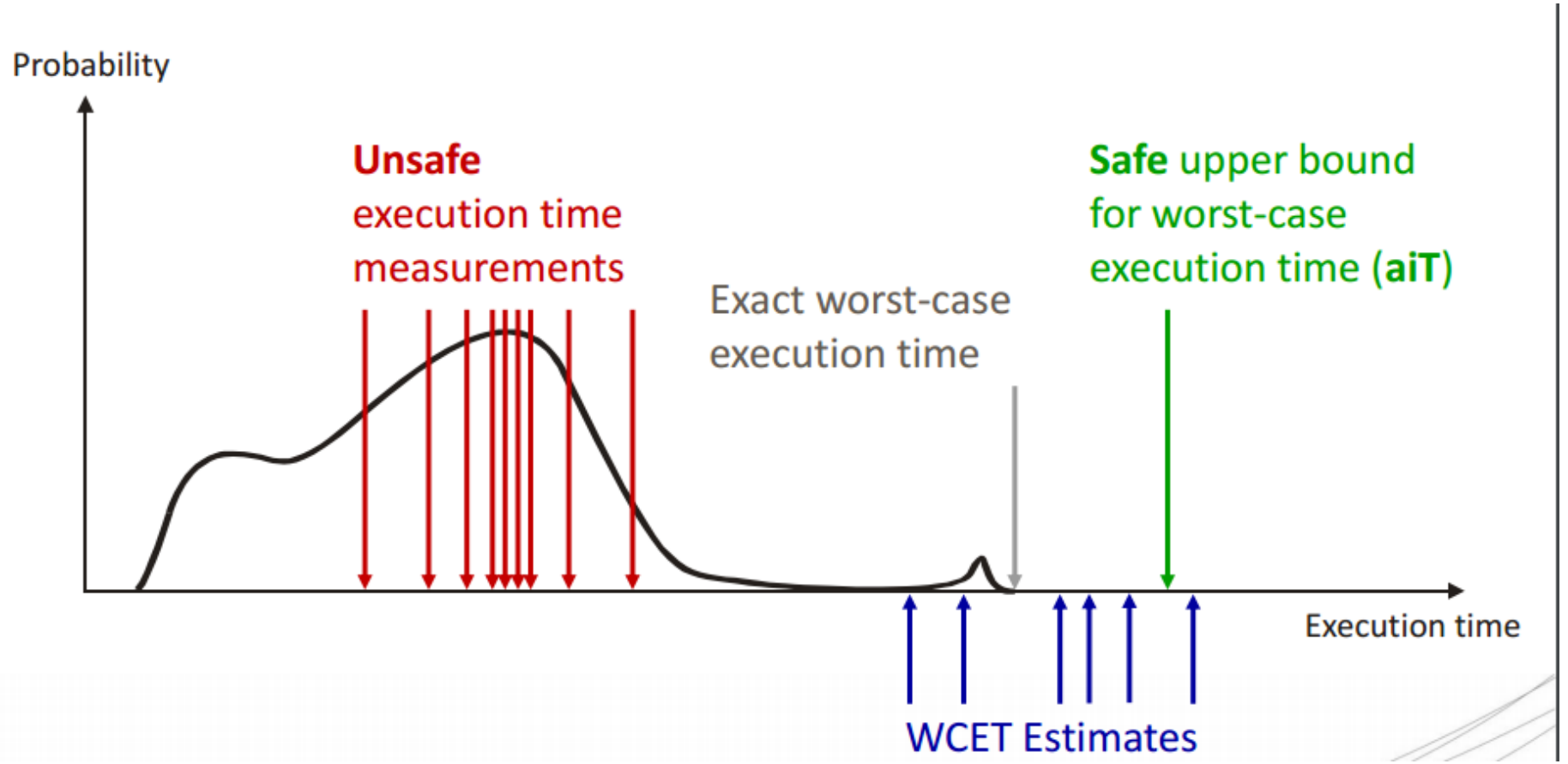
- Running code, measuring?

- No There are too many paths in non-trivial code

  e.g. **foo(unsigned a, unsigned b);**
  2^32 * 2^32 paths.

- **Static analysis** is the answer. Make use of ***abstract interpretation***

WCET using static-analysis



Why connect LLVM with a WCET tool?

Why connect LLVM with a WCET tool?

1) Provide WCET analysis alongside compilation
2) Re-use information about architecture (i.e. TableGen)

- Why not add WCET analysis to LLVM, instead of tool?
  - WCET analysis is not easy (Abstract Interpretation)
  - Why do the same work again?
  - To test if LLVM has enough info in TableGen.

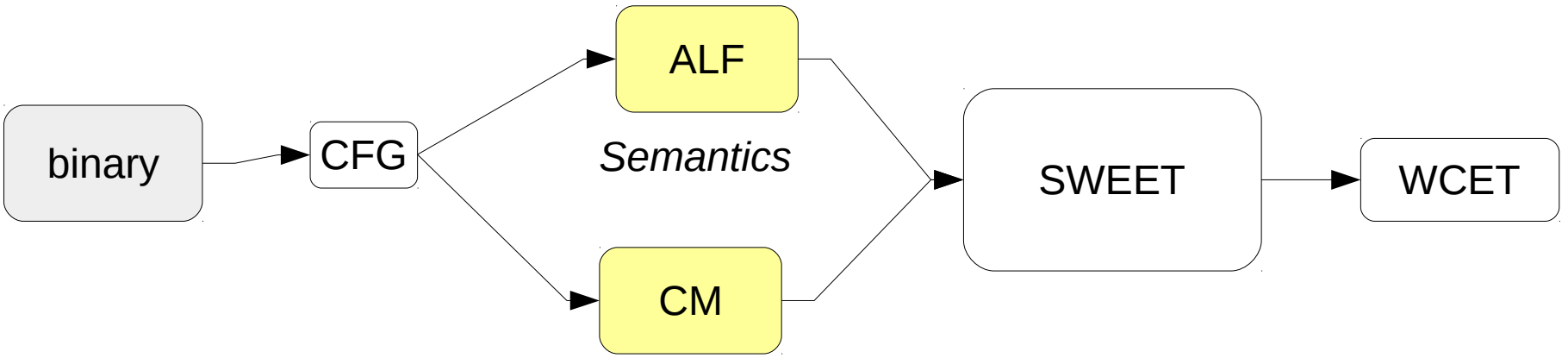
What tool to pick? (SWEET)

What tool to pick? (SWEET)

- SWEdish Execution Time
- Open-source tool
- Has interface language ALF
- Other tools considered:
  Bound-T, OTAWA, Hepatane, Absint aiT

How to use SWEET

- SWEET requires *cycle model* and *semantics*
  - Cycle Model: How many cycles is this basic-block?
  - Semantics : Register changes, control flow etc



Example of ALF code

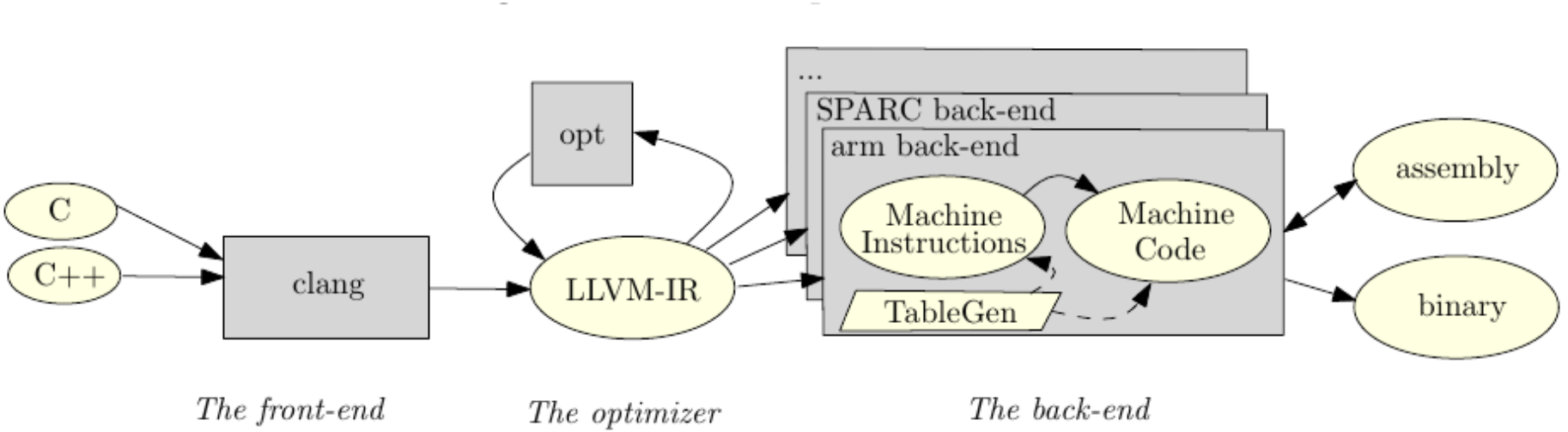
```
/*
%R1<def>, %CPSR<def,dead> = tADDrr %R0, %R1<kill>, pred:14, pred:%noreg;
*/
{ label 32 { lref 32 "des:BB#2:11" } { dec_unsigned 32 0 } }
{ store { addr 32 { fref 32 "R1" } { dec_unsigned 32 0 } } with
 { add 32
  { load 32 { addr 32 { fref 32 "R0" } { dec_unsigned 32 0 } } }
  { load 32 { addr 32 { fref 32 "R1" } { dec_unsigned 32 0 } } }
  { dec_unsigned 1 0 } }
}
```

Example of addition-instruction of two registers (R0, R1)

Approach to connecting SWEET and LLVM

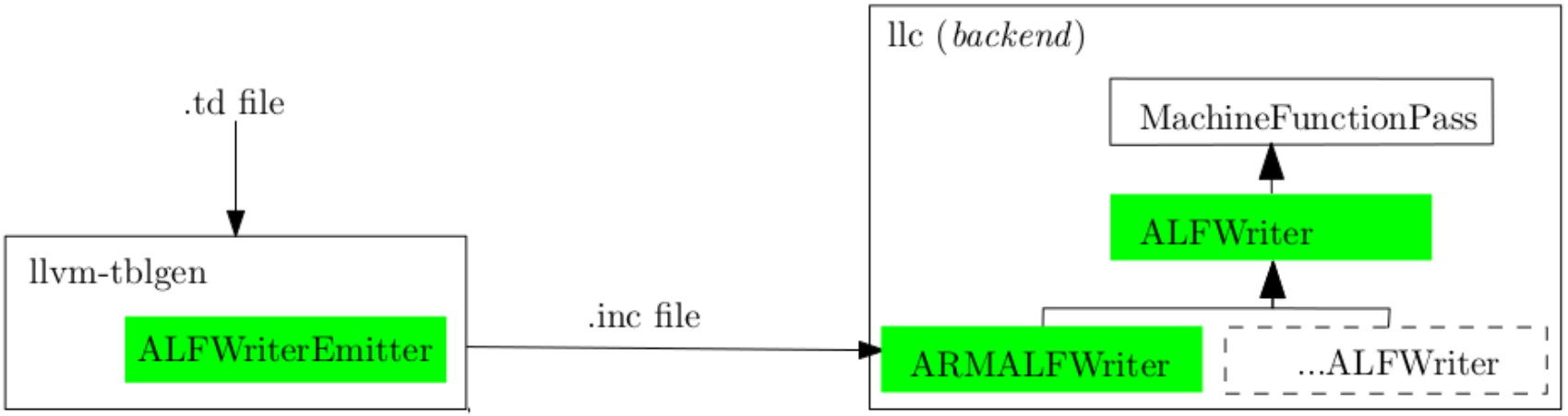
Approach to connecting SWEET and LLVM

- Output ALF from LLVM
- What ALF? Use info from TableGen
- Output from MI, just before conversion to MC. (addPreEmitPass of TargetPassConfig)



How to determine ALF per instruction

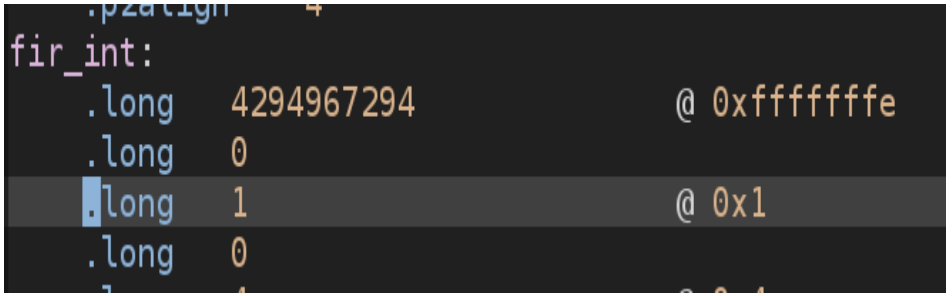
- TableGen back-end that generates ALF based on DAG-pattern
  - Generate function that determines ALF code for given *MachineInstr* object.
- *Most* instructions have a DAG-pattern
- Condition flags are assumed to be N, Z, C, V

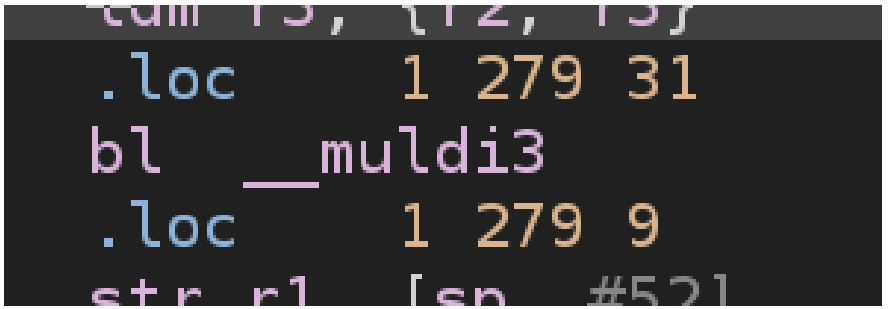


WCET for the ARM Cortex-M3

Generating for ARM Cortex-M3

- 52 of 86 Thumb1 covered (14 custom)
- Simple cycle model
- QEMU simulator used to test
- Discovered issues:
  1) Not all code available in LLVM: libgcc
  2) Globals are allocated by the linker, addr not known by LLVM





Conclusions

Conclusions

- Goal: Add WCET analysis with the SWEET tool
- SWEET runs on ALF code
- Using TableGen to generate ALF
  - Generated ALF for ARM Cortex-M3 for some programs
  - Condition flags not in TableGen
  - LLVM does not have all information (libgcc, globals)

Future work

- TableGen back-end uses fixed DAG-patterns
  e.g. (set .. (add … …))
- Instruction delay-slots not considered
- ARM Cortex-M3 instructions only partially finished
- Hand-write libgcc functions in ALF

Some links

- Code of LLVM with WCET additions
  https://github.com/rveens/LLVM-WCET-SWEET
- Vim syntax highlighting for ALF
  https://github.com/rveens/alf-vim
- SWEET homepage
  http://www.mrtc.mdh.se/projects/wcet/sweet/
- ALF spec
  http://www.es.mdh.se/pdf_publications/1138.pdf

end of presentation

How to determine ALF per instruction

- Considered patterns:

- Note:
  operators such as
  add, sub, ld etc
  are defined in
  TargetSelectionDAG.td

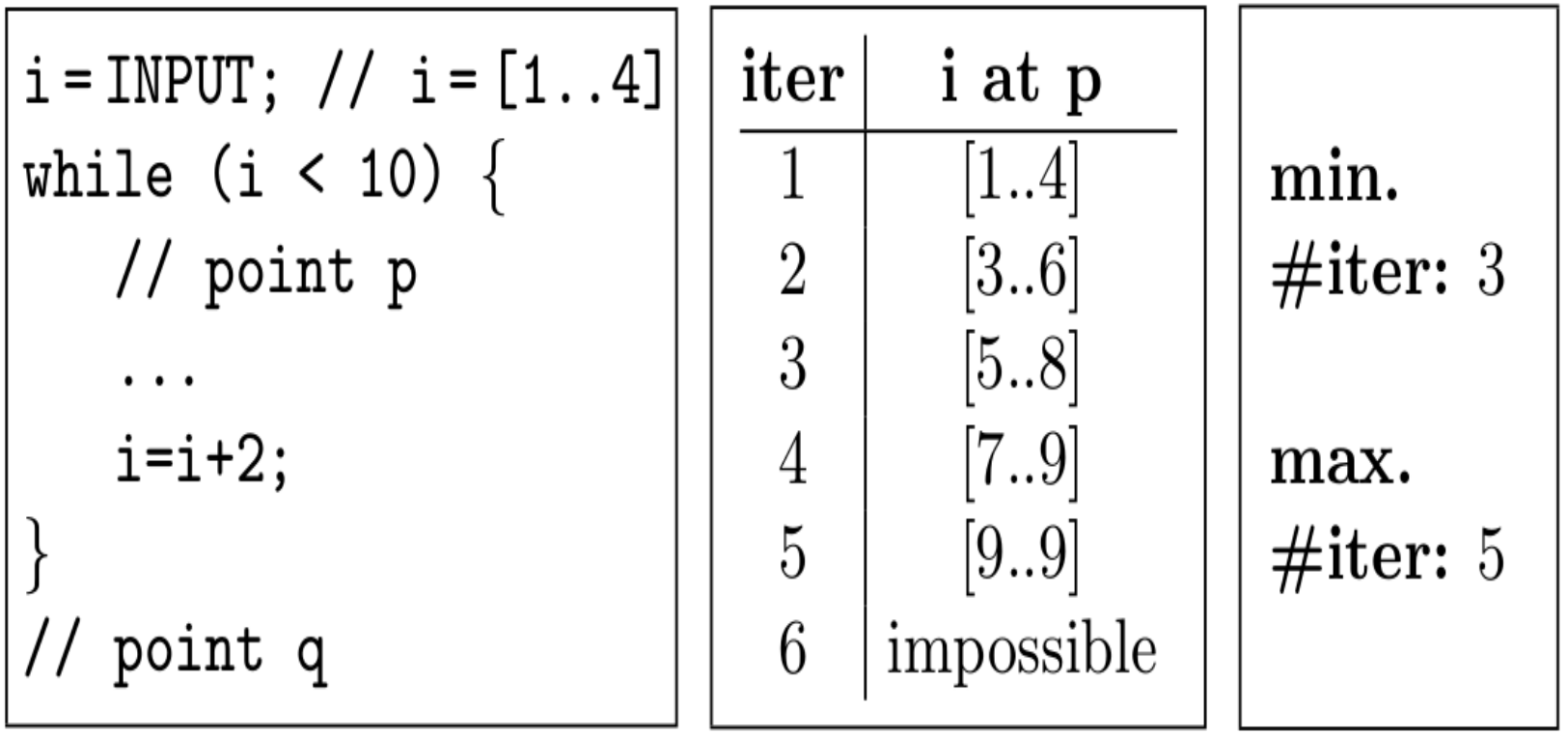
DAG-pattern	Flags set
(set $R (imm $L))	N, Z
(set $R (add $L, $L))	N, Z, C, V
(set $R (adde $L, $L))	N, Z, C, V
(set $R (sub $L, $L))	N, Z, C, V
(set $R (ld $L))	N, Z
(set $R (shl $L $L))	N, Z, V
(set $R (srl $L $L))	N, Z, V
(set $R (mul $L $L))	N, Z, V
(set $R (xor $L $L))	N, Z, V
(set $R (or $L $L))	N, Z, V
(set $R (and $L $L))	N, Z, V
(set $R (sext_inreg $L $Type))	N, Z
(st $L $L)	-
(br $BB)	-

LLVM TableGen example

```
def tADC :                          // A8.6.2
  T1sItDPEncode<0b0101, (outs tGPR:$Rdn), (ins tGPR:$Rn, tGPR:
$Rm), IIC_iALUr,
                  "adc", "\t$Rdn, $Rm",
                  [(set tGPR:$Rdn, (adde tGPR:$Rn, tGPR:$Rm))]>,
Sched<[WriteALU]>;
```

```
def tADC {      // Instruction InstTemplate InstThumb
….T1sItDPEncode Sched
 field bits<32> Inst = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 1, 0, 1, ... };
 dag OutOperandList = (outs tGPR:$Rdn, s_cc_out:$s);
 dag InOperandList = (ins tGPR:$Rn, tGPR:$Rm, pred:$p);
 string AsmString = "adc${s}${p}       $Rdn, $Rm";
 list<dag> Pattern = [(set tGPR:$Rdn, (adde tGPR:$Rn, tGPR:$Rm))];
 list<Register> Uses = [CPSR];
 list<Register> Defs = [];
```

SWEET Abstract Execution



```
i = INPUT; // i = [1..4]
while (i < 10) {
   // point p
   ...
   i=i+2;
}
// point q
```

(a) Example

iter	i at p
1	[1..4]
2	[3..6]
3	[5..8]
4	[7..9]
5	[9..9]
6	impossible

(b) Analysis

min.
#iter: 3

max.
#iter: 5

(c) Result

Figure 1. Example of abstract execution