PEak + Mapper Update

Ross Daly

Outline

- Review what PEak is and our approach to technology mapping
- How well did this approach work in regards to the tapeout
- What new features do we want (need) to add to get a full working system which can do design space exploration

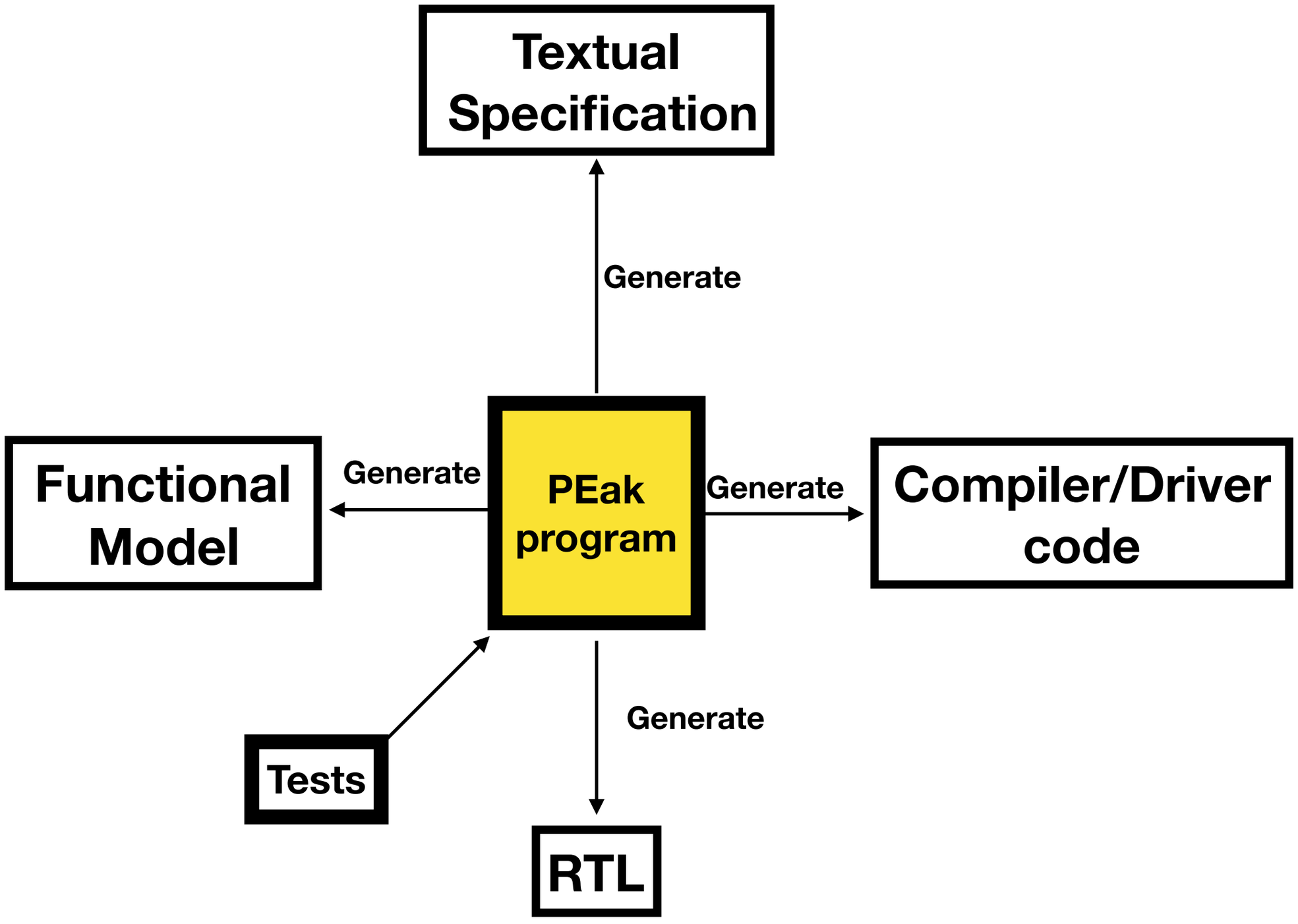
Goal:

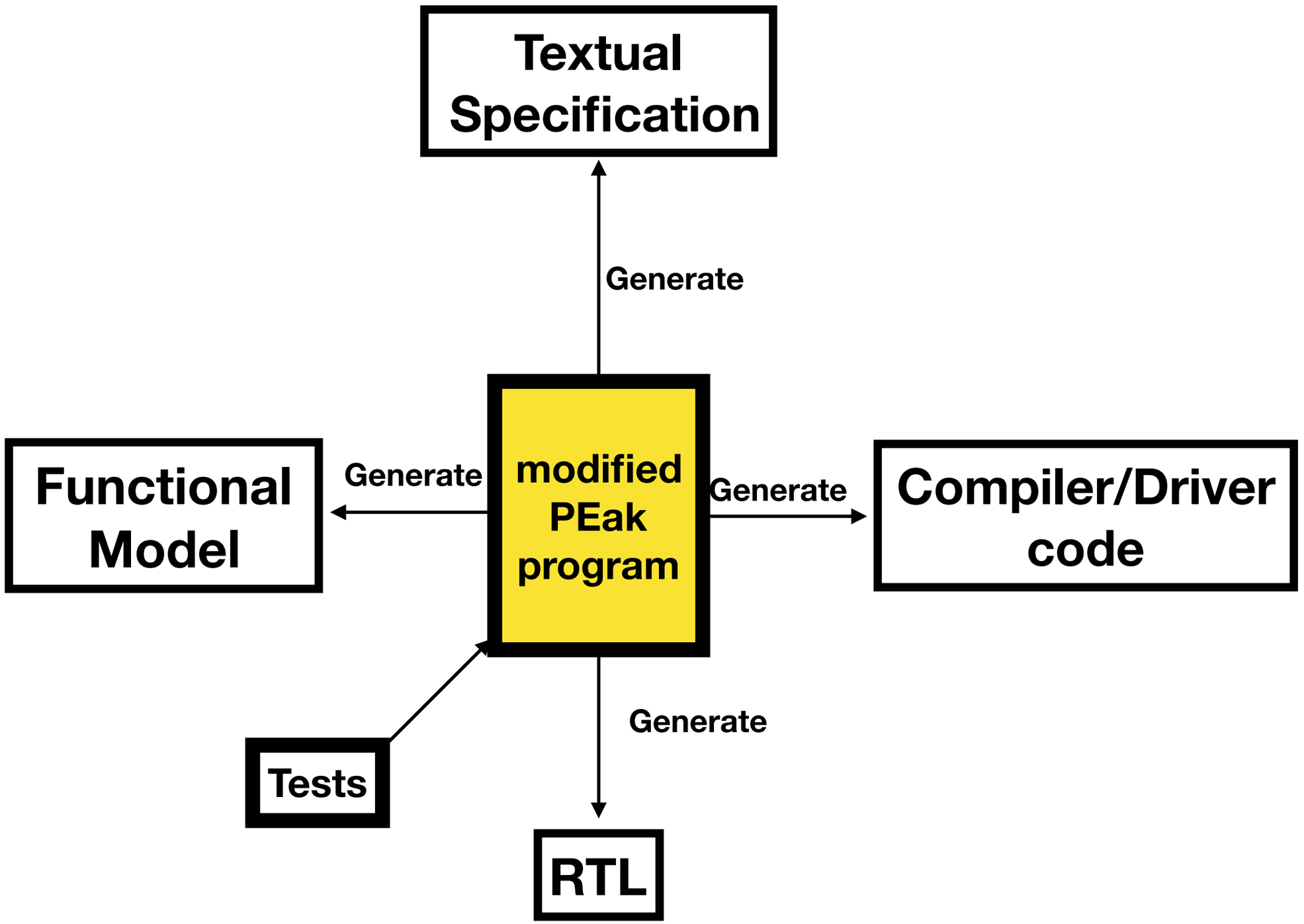
Build tools that enable the agile philosophy

“Traditional hardware flows”

- Often multiple sources of truth that are in conflict with each other
    - Source of lots of debugging man-hours
    - Different tests for each “source of truth”
- Even small changes to the specification can require large manual rewrites of the software/compiler stack
    - Adding Features/Fixing Bugs requires changes in many places
    - Design Space Exploration is difficult/time consuming
- Software was always second class priority.

PEak: The single source of truth

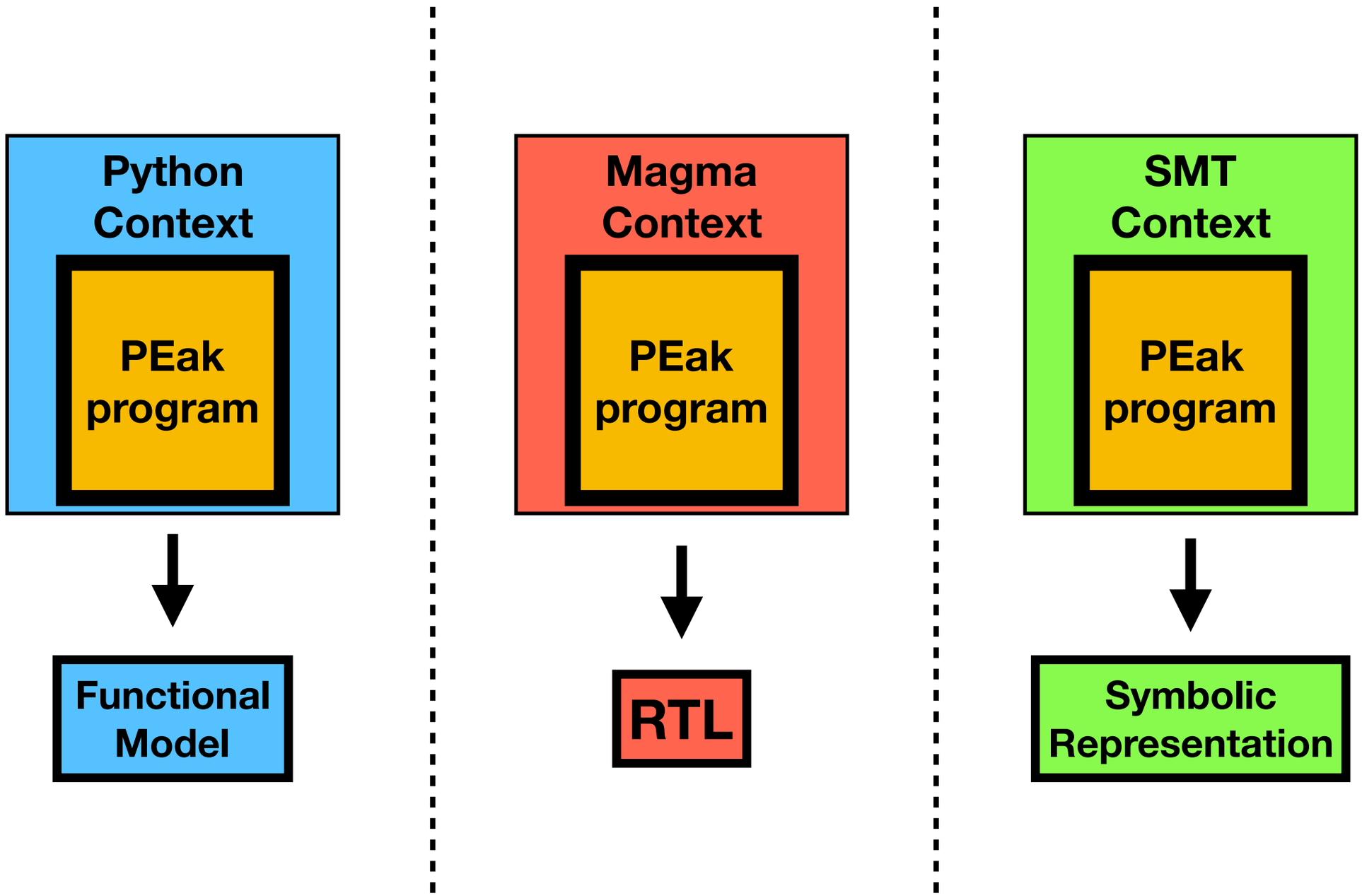

Textual Specification
Generate
Functional Model
Generate
PEak program
Generate
Compiler/Driver code
Tests
Generate
RTL


Textual Specification
Generate
Functional Model
Generate
modified PEak program
Generate
Compiler/Driver code
Tests
Generate
RTL

PEak

- DSL Used to specify our Processing Elements in the CGRA
  - Instruction Set
  - Precise semantics of instructions
  - PEak program should be able to generate:
    - Functional Model
    - Hardware Description (RTL)
    - Rewrite rules for Compiler (Technology mapping)
    - Assembler/Dissasembler (Bitstream for configured PE tiles)
- Key Idea: Single program, multiple interpretation

Multiple Interpretations



What is a PEak program?

- Defines a specific Instruction Set
- Declares all state (Similar to ISP)
- Precisely describes instruction semantics

Describing the Instruction Set

- Algebraic Data types
    - Product Types (Tuples, Records/Structs)
        - typeA **and** typeB **and** TypeC
    - Sum Types (Variants, Enums/Tagged Unions)
        - typeA **or** typeB **or** TypeC

Describing PE semantics

isa.py

```
class ALUOP(Enum):
    Add = 0
    Or  = 1
    And = 2
    XOr = 3
    #etc...

class Instr(Product):
    alu_op : ALUOP
    invert_in0 : Bit
    invert_in1 : Bit
    set_carry : Bit
```

semantics.py

```
ABV = hwtypes.AbstractBitVector
class SimplePE(Peak):
    def __init__(self):
        #Declare all the state

    def __call__(self, instr : Instr,
                       in0 : ABV[16],
                       in1 : ABV[16]):
        if instr.invert_in0:
            in0 = ~in0
        if instr.invert_in1:
            in1 = ~in1

        carry = instr.set_carry
        if instr.op == Op.Add:
            res = in0 + in1 + carry
        elif instr.op == OP.And:
            res = in0 & in1
        #etc…
        return res
```

Assembler

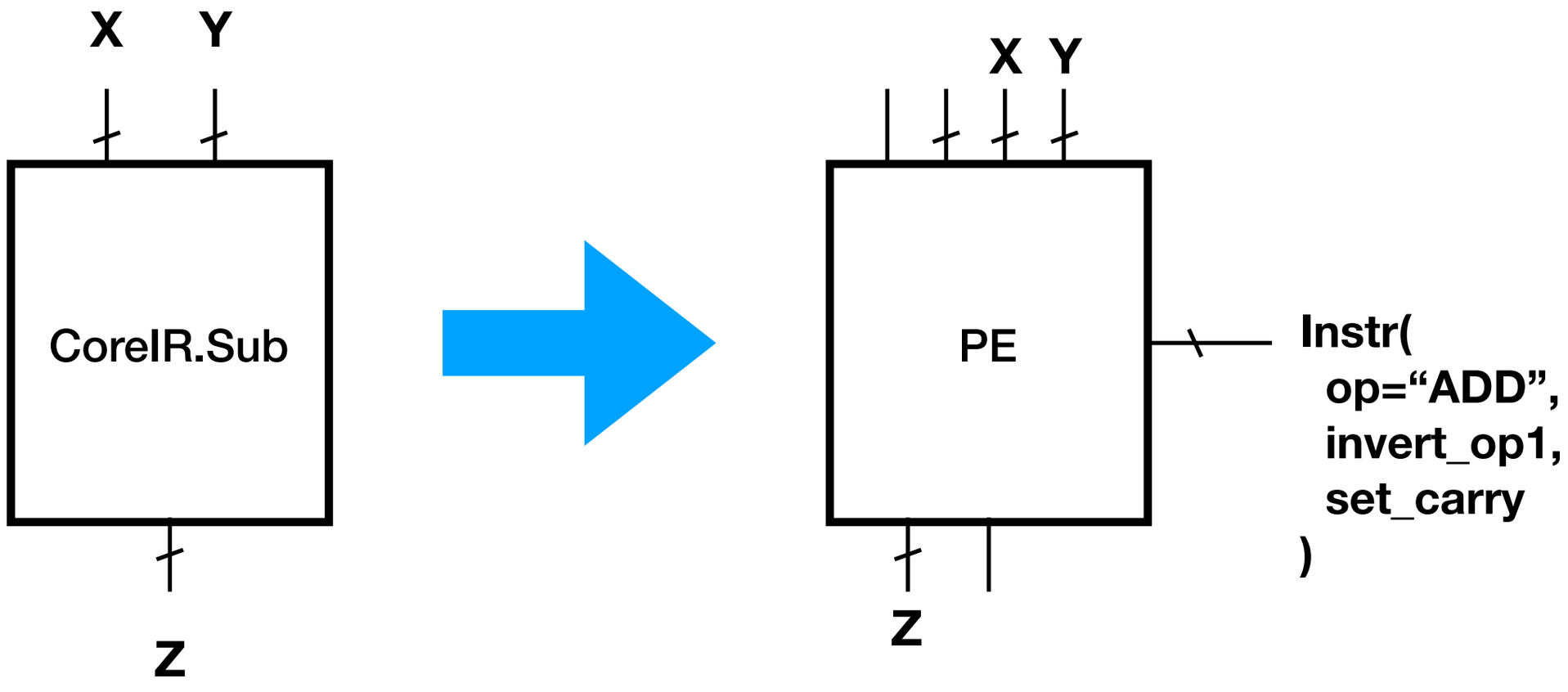
```
class ALUOP(Enum): #Sum Type
    Add = 0
    Or  = 1
    And = 2
    XOr = 3
#etc...

class Instr(Product): #Product Type
    alu_op : ALUOP
    invert_in0 : Bit
    invert_in1 : Bit
    set_carry : Bit


assembler = gen_assembler(Instr)
dissassembler = gen_dissasembler(Instr)

sub_instr = Instr(ALUOP.Add,0,1,1)
assembled = assembler(sub_instr)
assert sub_instr == dissassembler(assembled)
```

Technology Mapping

- Foreach primitive in CoreIR:
    - Manually write one (or more) **Rewrite Rule** from that primitive to a PE instruction (and an interface binding)

- Given an application graph of CoreIR primitives
    - Foreach node in application graph:
        - Apply best rewrite rule to generate PE graph (Instruction Selection)

‘Sub’ Rewrite Rule



Algorithm for Automatically Generating Rewrite Rule from PEak

```
foreach coreir primitive:
  coreir_formula = primitive.smt_formula
  foreach PEak instruction in Instr:
    pe_formula = pe(instr,smtBV,smtBV)
    if coreir_smt is equivalent to pe_smt:
      Found a rewrite rule for the primitive!
```

Tapeout: What went well?

- Compared to last year… A lot!
- Conceptual “Single source of truth” was a success
    - PEak description
        - Formally defined semantics of operations (BitVector + Floating Point)
- Mapping rewrite rules were generated!
- ADT abstraction + Assembler/Disassembler were easy to use.
- Testing/Verification *significantly* improved.
    - Same tests for functional and RTL!

Tapeout: What Needs Improvement?

Where is the human in the loop?

- Automapper did not solve the *complete* compilation problem
- Memory compilation was manual and hacky
- Python vs C++ rewrite rules
- Registers were special cased
- Did not do packing

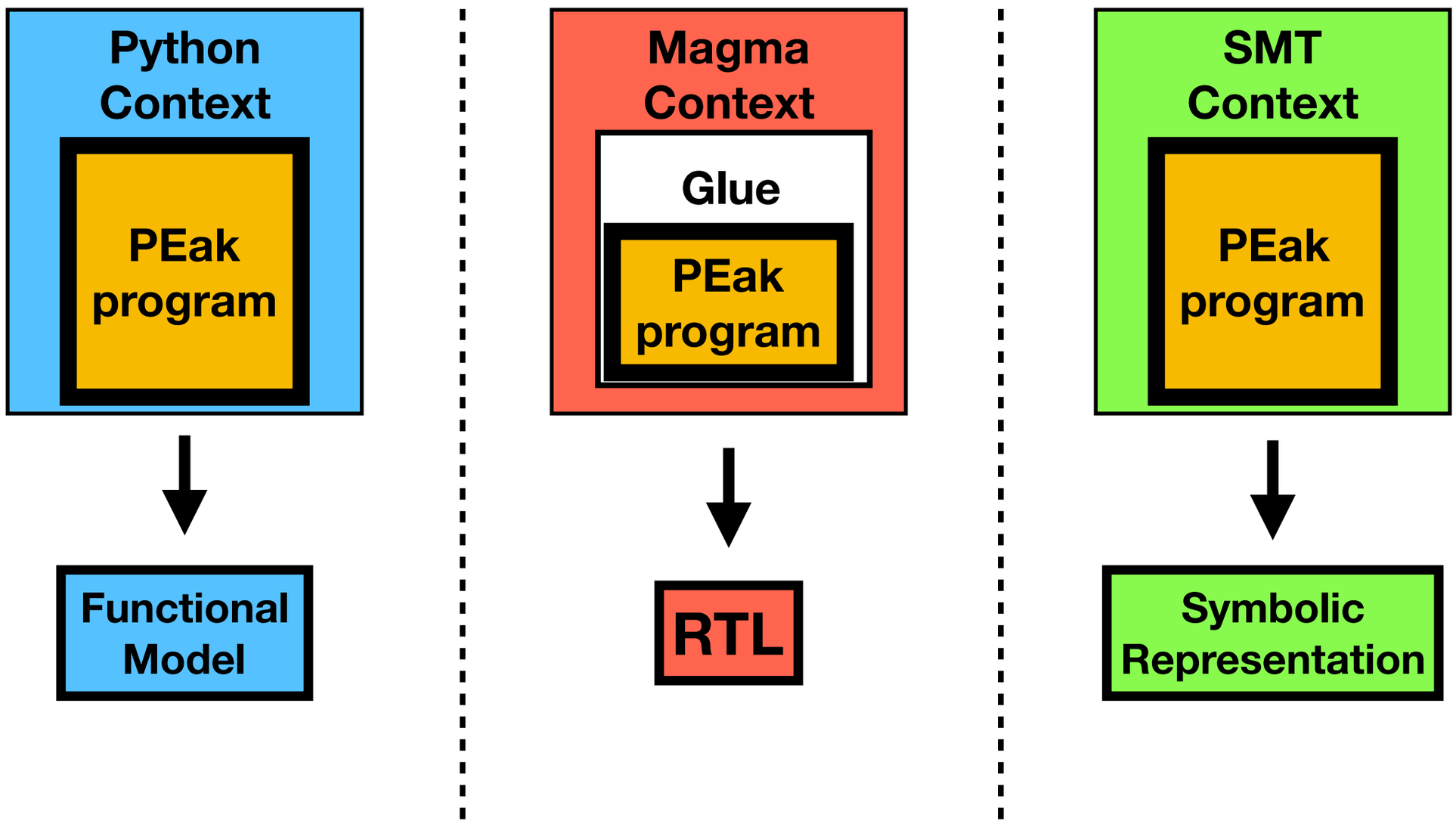
Where is the human in the loop?

- Packing was put on the back burner…
    - Register and Constant packing were absolutely necessary to fit many Apps
    - Missing LUT packing
    - Should have mapped to simpler primitives
- PEak RTL was not easy to edit using Garnet
    - Manual addition of register reading/writing logic in PEak

Quality/Performance Issues

- Quality of generated RTL from PEak
    - Circuit reuse (Multiplier)
    - Structural verilog vs always blocks
- Deriving all the rules is slow
    - Enumerating instructions in Python vs using SMT
    - Mapping to Full PE
- Instruction Selection
    - Only tried to find single rewrite rule for each coreir primitive
    - Did not attempt to find any fusion opportunities (FMA)

Minor issues



Future: Design Space Exploration

- We can use full power of python to meta-program PEak programs!

- foreach parameterization of the PEak spec:
    - run full halide benchmark suite and measure performance/area/timing/etc

- Can do massive searches given that there is no human in the loop

    - We are almost there.

Thank you!