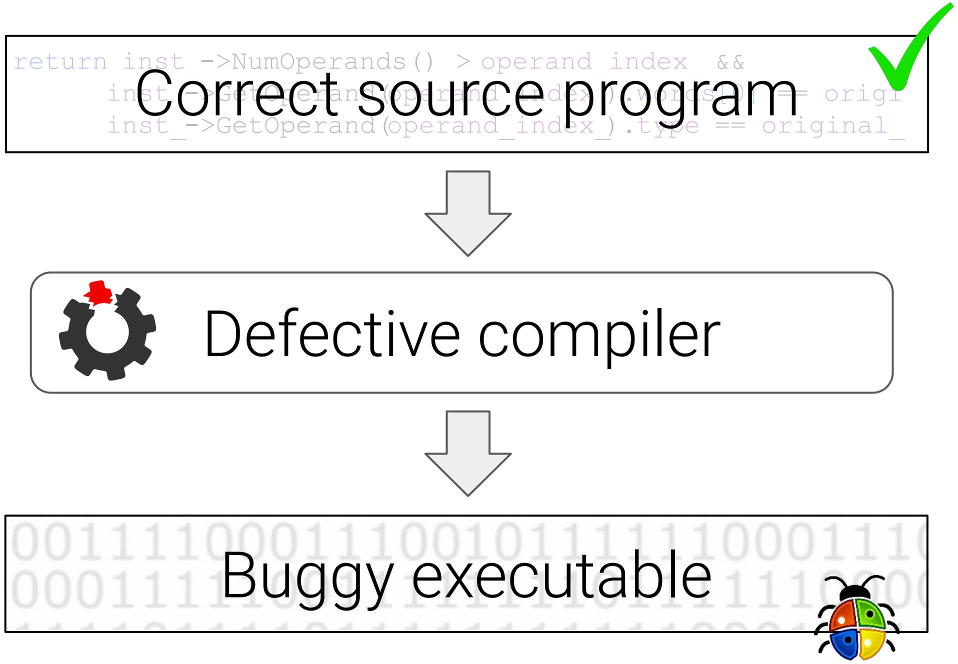
Low Fat Recipes for Reliable Programming Languages

Alastair F. Donaldson, Imperial College London, UK

Huawei Global Technology Summit, July 2022

Programming language implementations need to be reliable!



Correct source program ✓

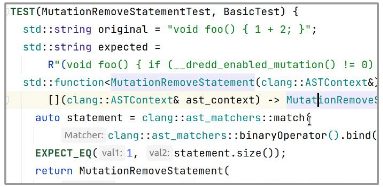
Defective compiler

Buggy executable

return inst ->NumOperands() > operand index &&
inst_->GetOperand(operand_index_).type == original_
00111100011100101111110001110

Programming language specifications need to be clear!

Programmer



```
TEST(MutationRemoveStatementTest, BasicTest) {
  std::string original = "void foo() { 1 + 2; }";
  std::string expected =
      R"(void foo() { if (__dredd_enabled_mutation() != 0)
  std::function<MutationRemoveStatement(clang::ASTContext&)
      [](clang::ASTContext& ast_context) -> MutationRemoveS
    auto statement = clang::ast_matchers::match(
        Matcher: clang::ast_matchers::binaryOperator().bind(
    EXPECT_EQ( val1: 1,  val2: statement.size());
    return MutationRemoveStatement(
```

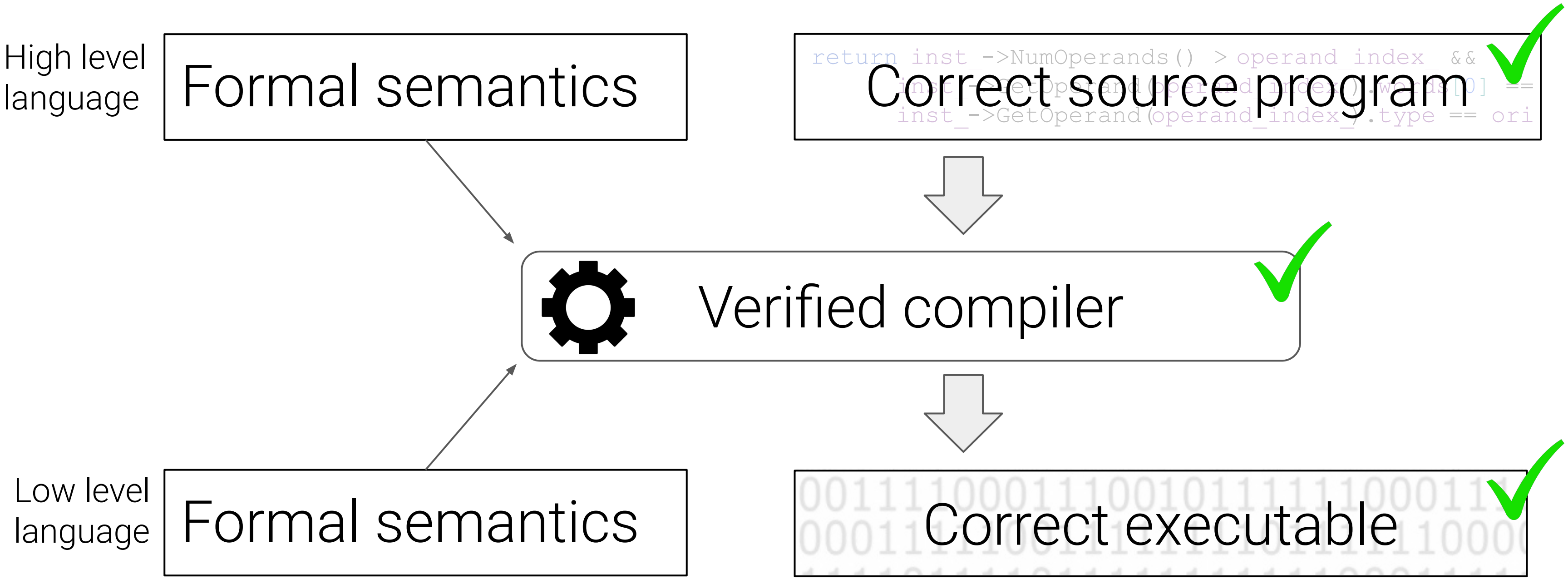
Language spec

Compiler

Verifier

Compiler

The dream: mechanised programming languages and tools



High level language

Formal semantics

Correct source program ✓

Verified compiler ✓

Low level language

Formal semantics

Correct executable ✓

return inst_->NumOperands() > operand_index_ &&
inst_->GetOperand(operand_index_).type == ori
0011110001110010111111000111
0001111100111111101111110000

The reality

- New languages in state of flux
- Not enough expertise in formal semantics
- Verified compilers are expensive to create
- Verified compilers are expensive to maintain

Low fat recipes for reliable programming languages

Let us look at **pragmatic** approaches for making programming languages more reliable

- Randomized testing
- Lightweight formal methods

Part 1: Randomized testing

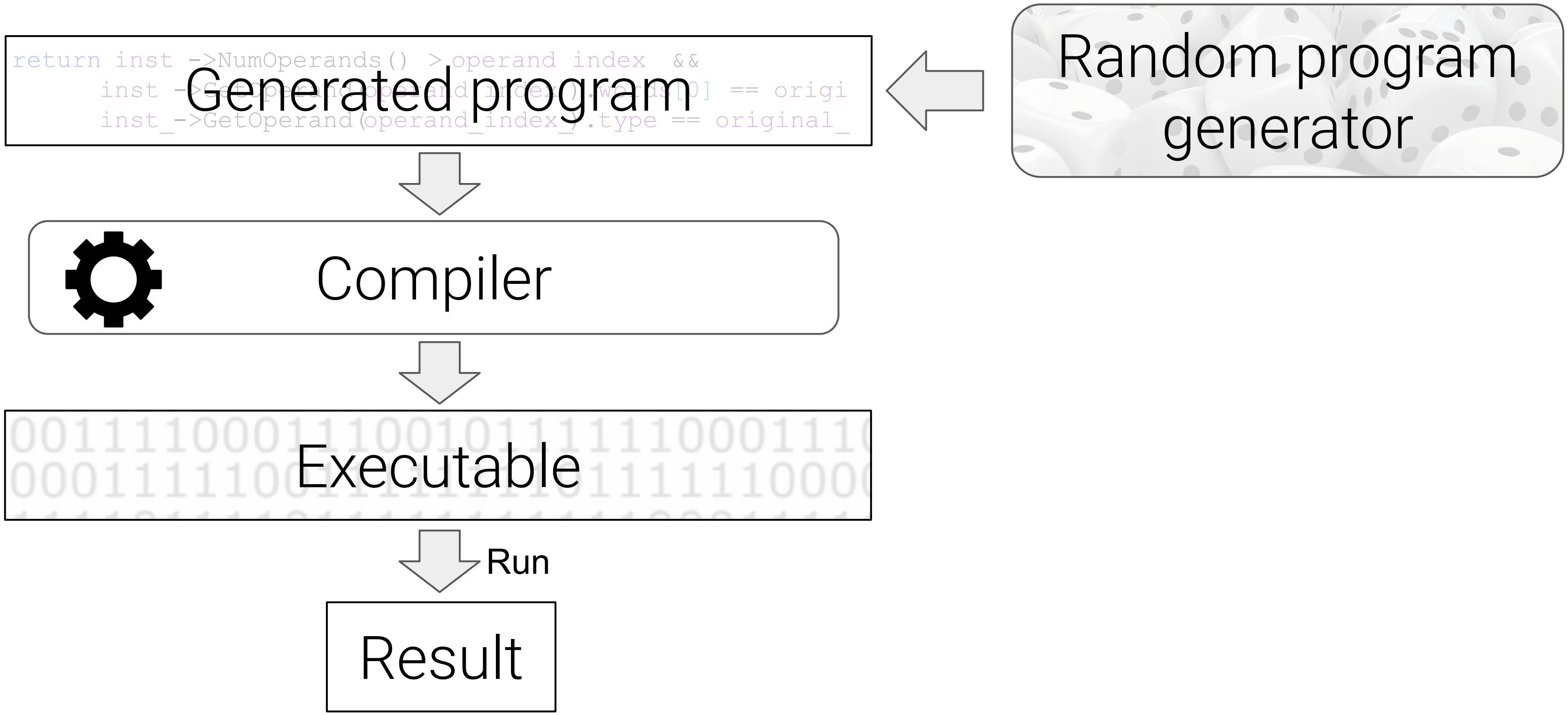
Generate random programs

Check that compilers do the right thing with them

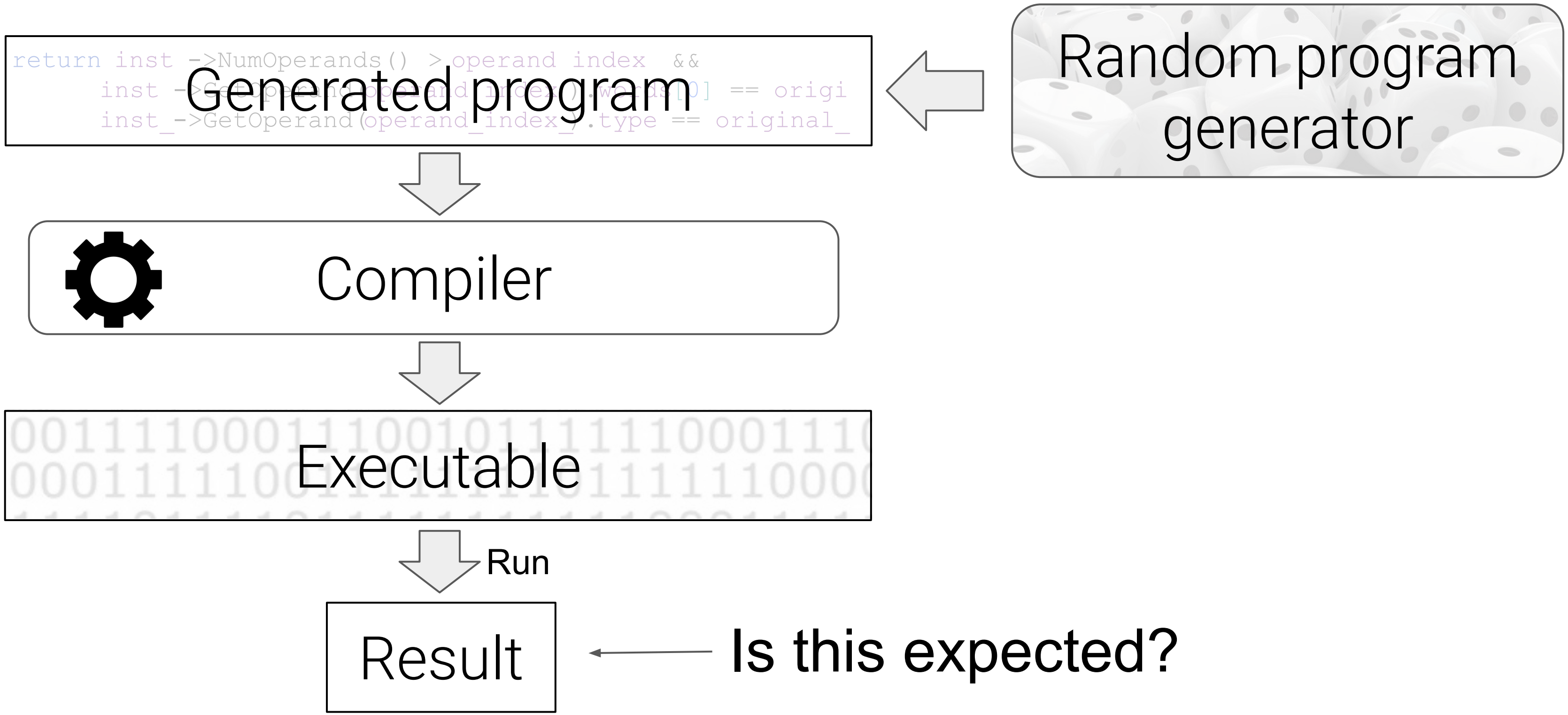
Great for finding bugs!

Great for highlighting murky corners of the language

Appealing idea: randomized testing for compilers



But … the *oracle problem* for compiler testing is hard!



Random program generator

Generated program

Compiler

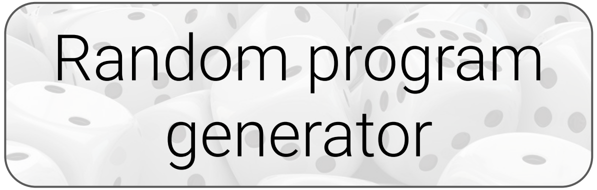
Executable

Run

Result

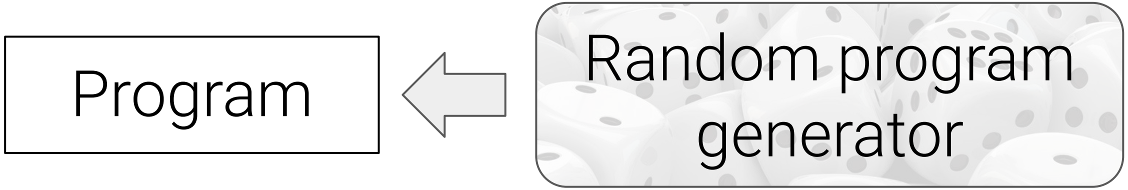
Is this expected?

Pseudo-oracle: differential testing



Random program generator

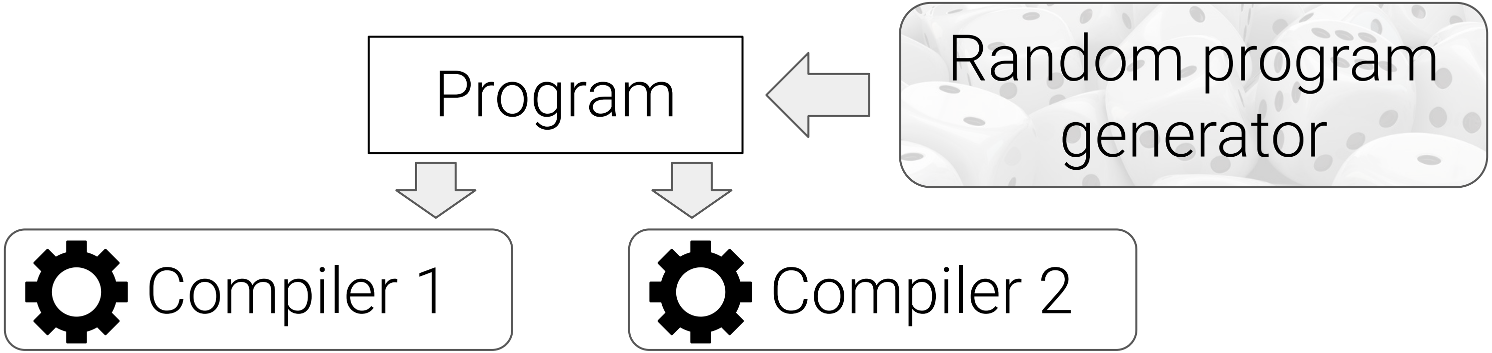
Pseudo-oracle: differential testing



Program

Random program generator

Pseudo-oracle: differential testing



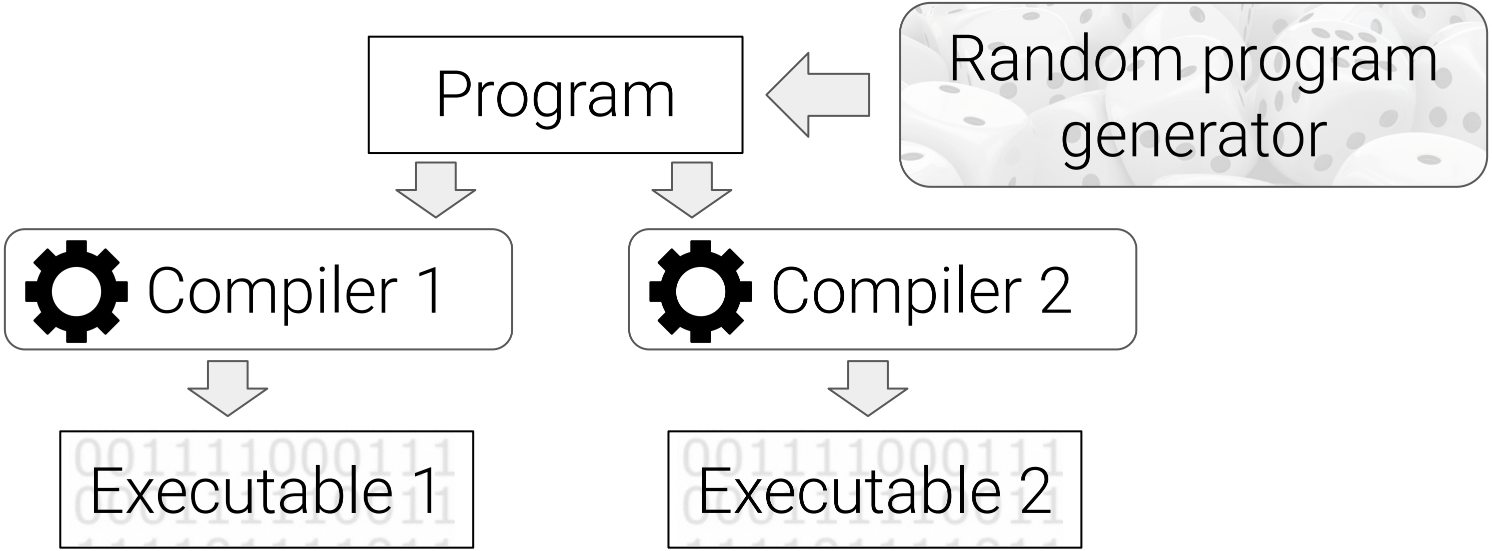
Program

Random program generator

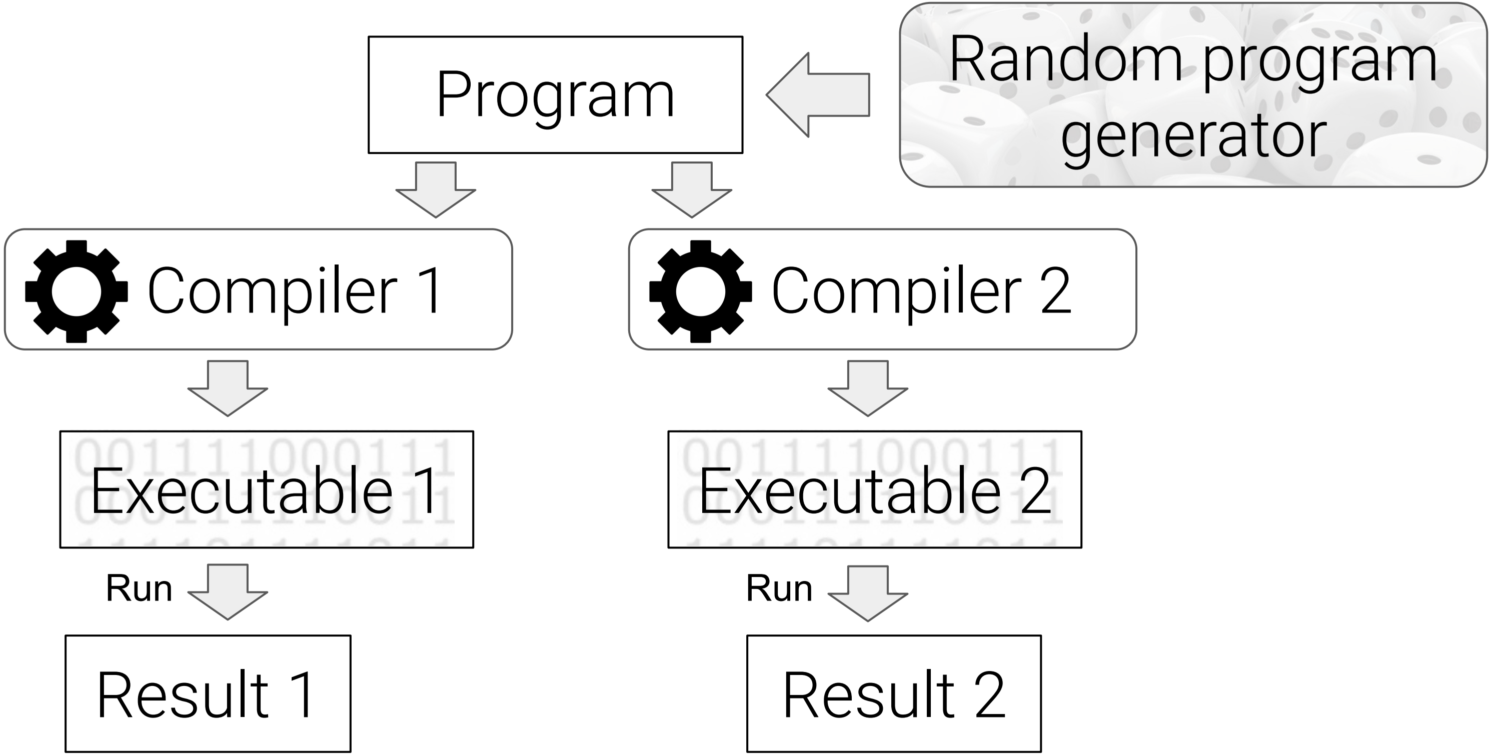
Compiler 1

Compiler 2

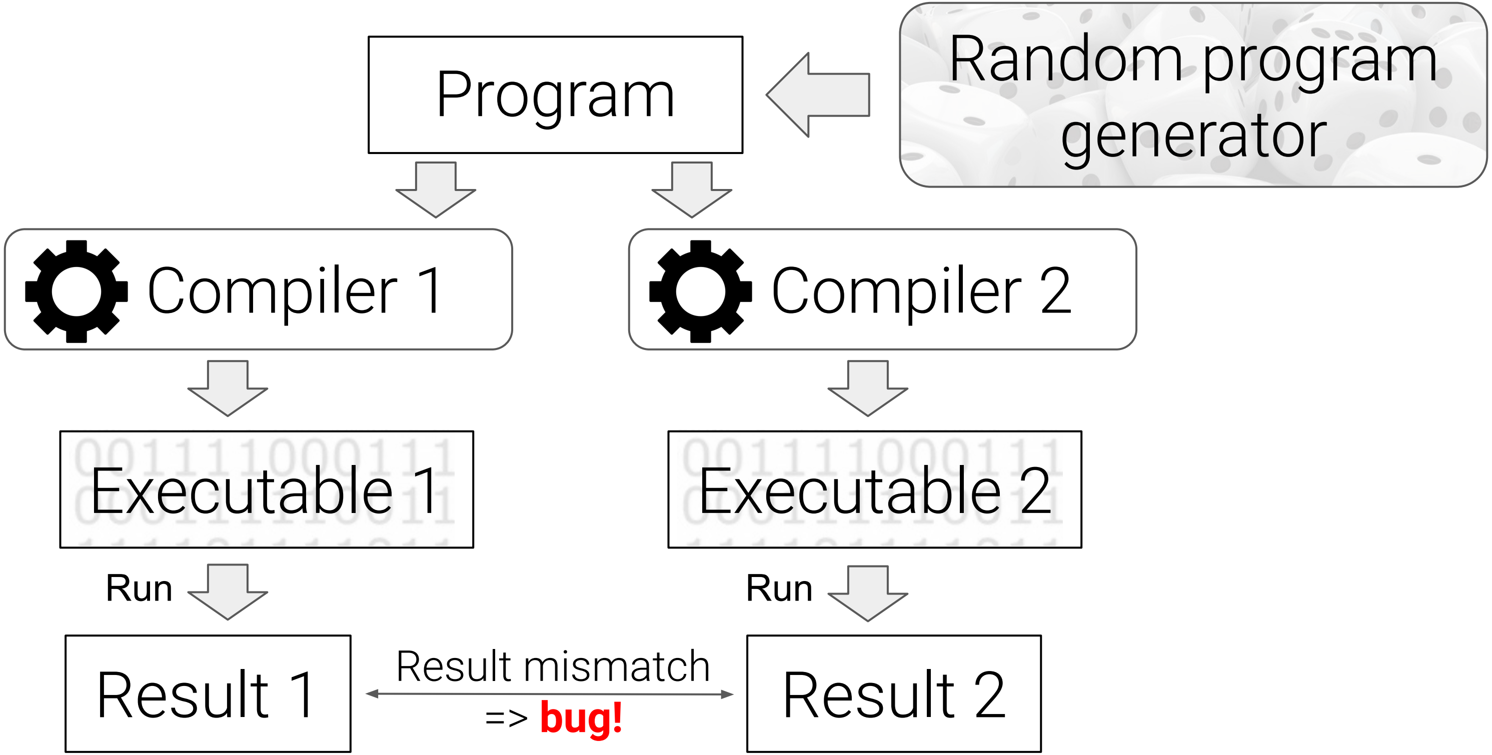
Pseudo-oracle: differential testing



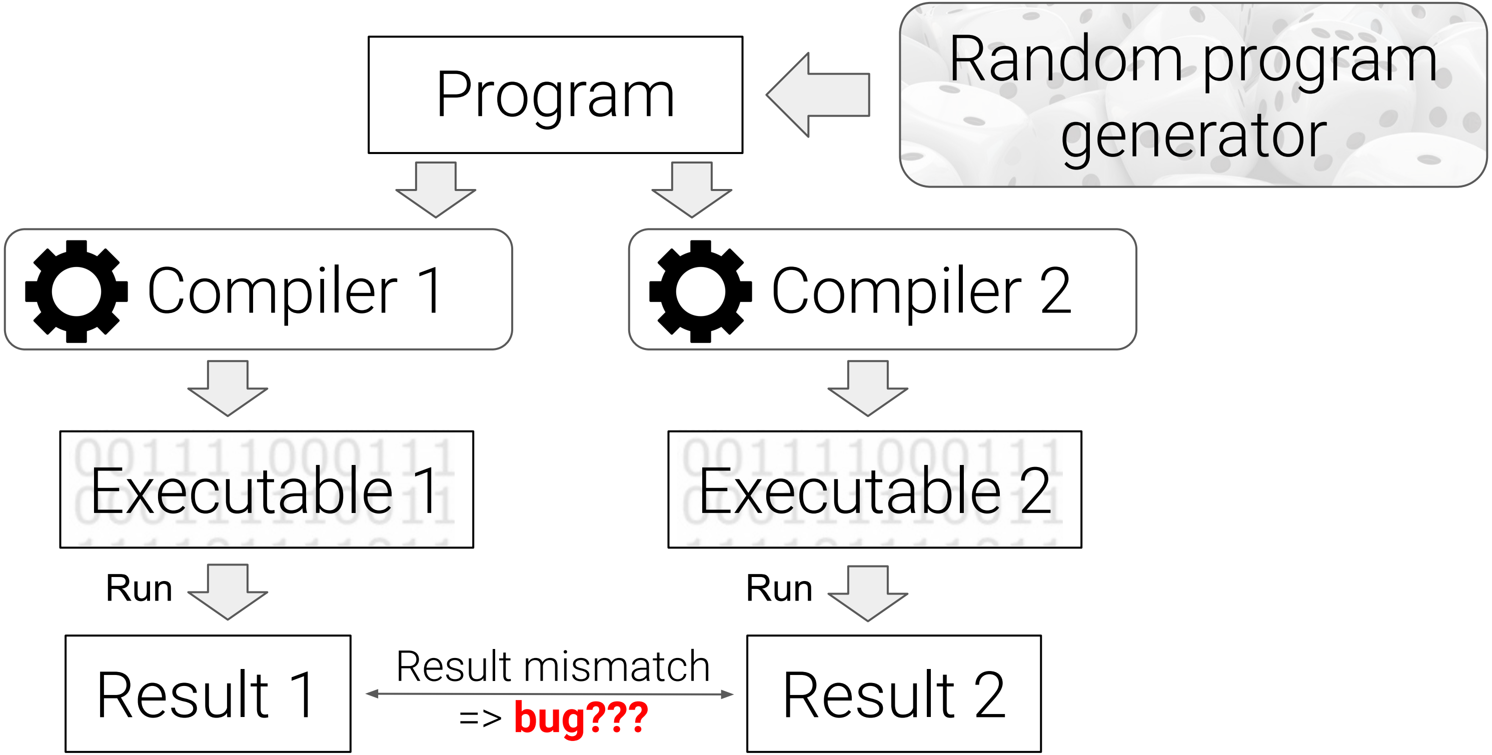
Pseudo-oracle: differential testing



Pseudo-oracle: differential testing



Pseudo-oracle: differential testing



Random program generator

Program

Compiler 1

Compiler 2

Executable 1

Executable 2

Run

Run

Result 1

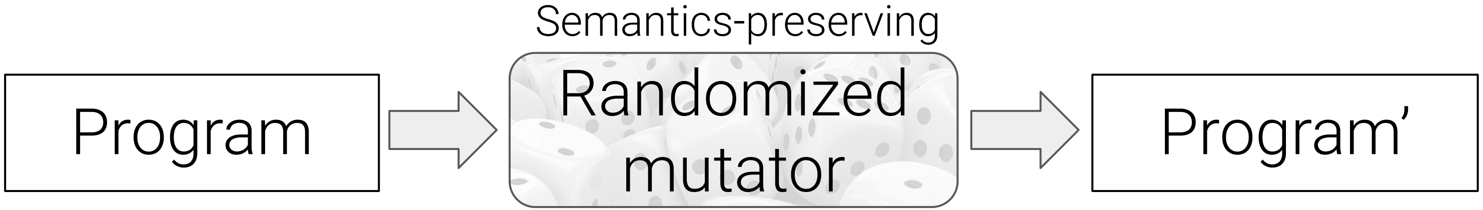
Result 2

Result mismatch

=> **bug???**

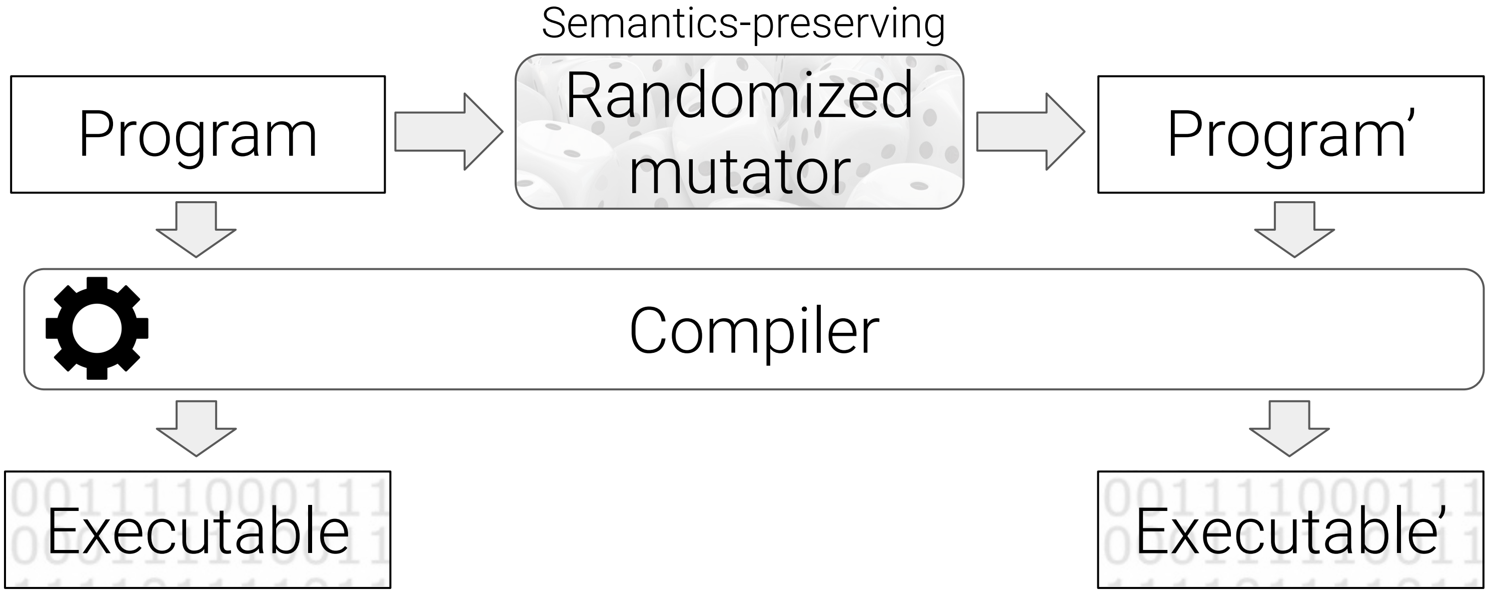
Le et. al, PLDI’14
Donaldson et al., OOPSLA’17

Pseudo-oracle: metamorphic testing



Le et. al, PLDI'14
Donaldson et al., OOPSLA'17

Pseudo-oracle: metamorphic testing



Semantics-preserving

Randomized mutator

Program

Program'

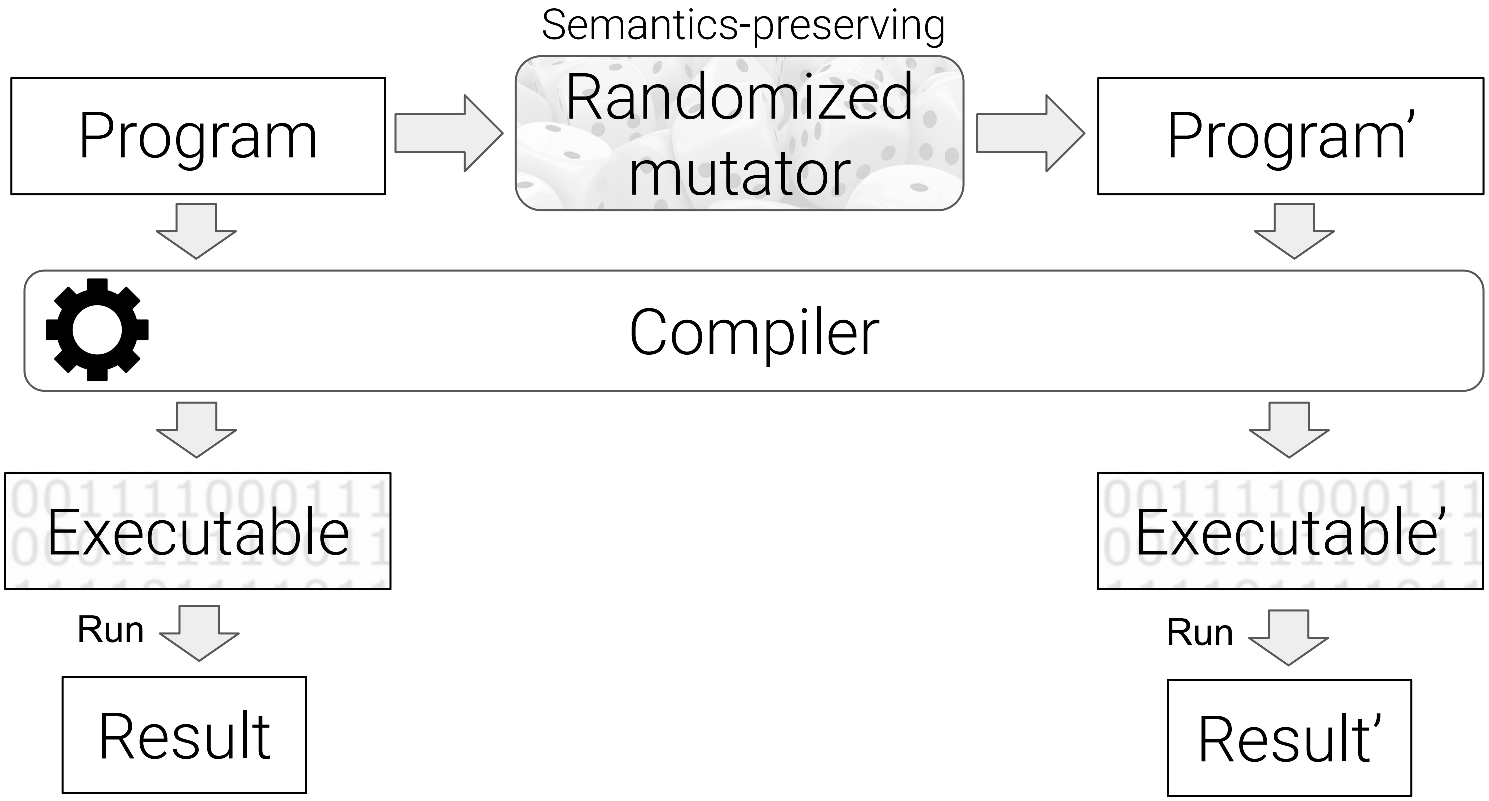
Compiler

Executable

Executable'

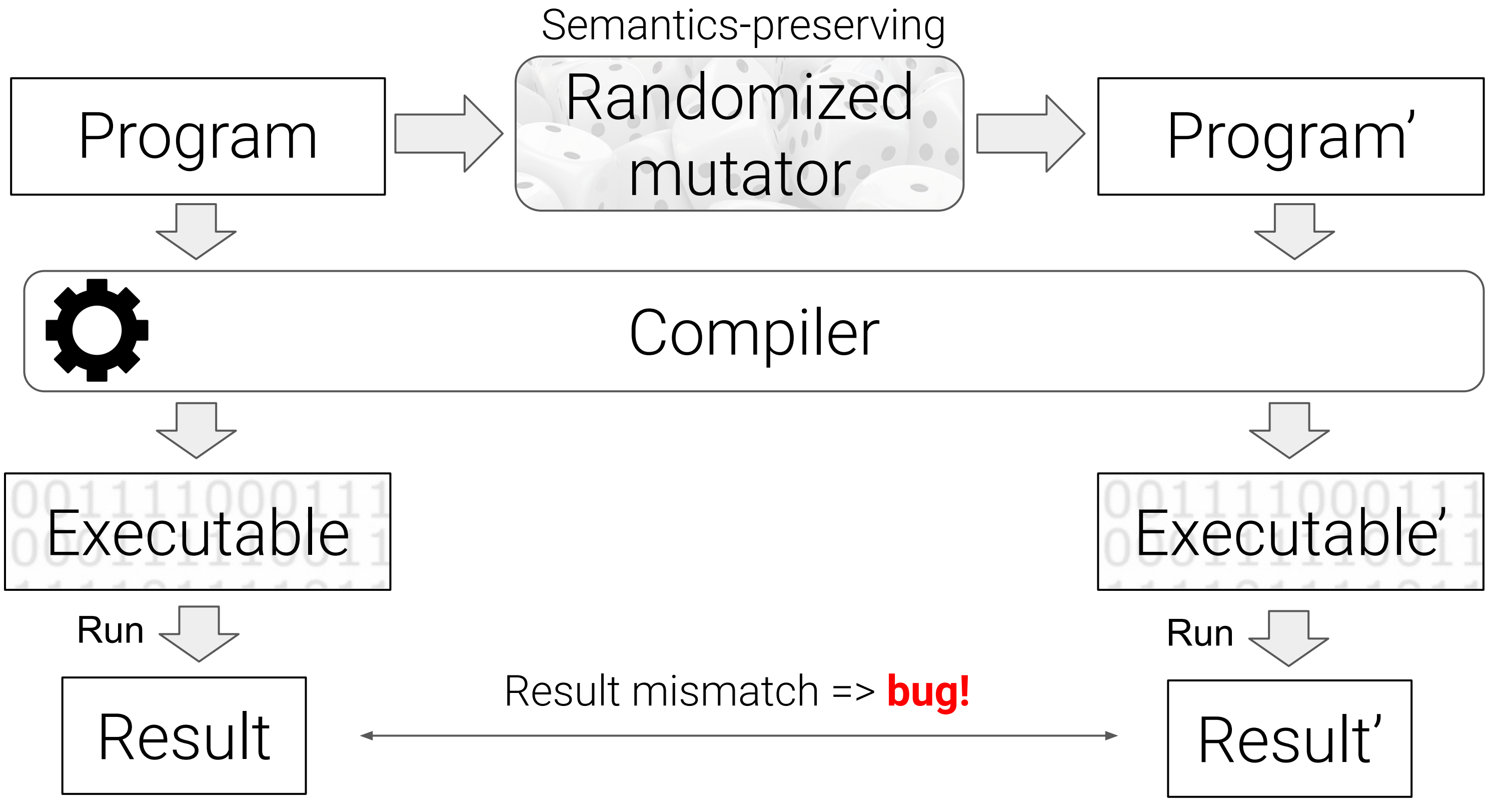
Le et. al, PLDI’14
Donaldson et al., OOPSLA’17

Pseudo-oracle: metamorphic testing



Le et. al, PLDI'14
Donaldson et al., OOPSLA'17

Pseudo-oracle: metamorphic testing



Semantics-preserving

Randomized mutator

Program

Program'

Compiler

Executable

Executable'

Run

Run

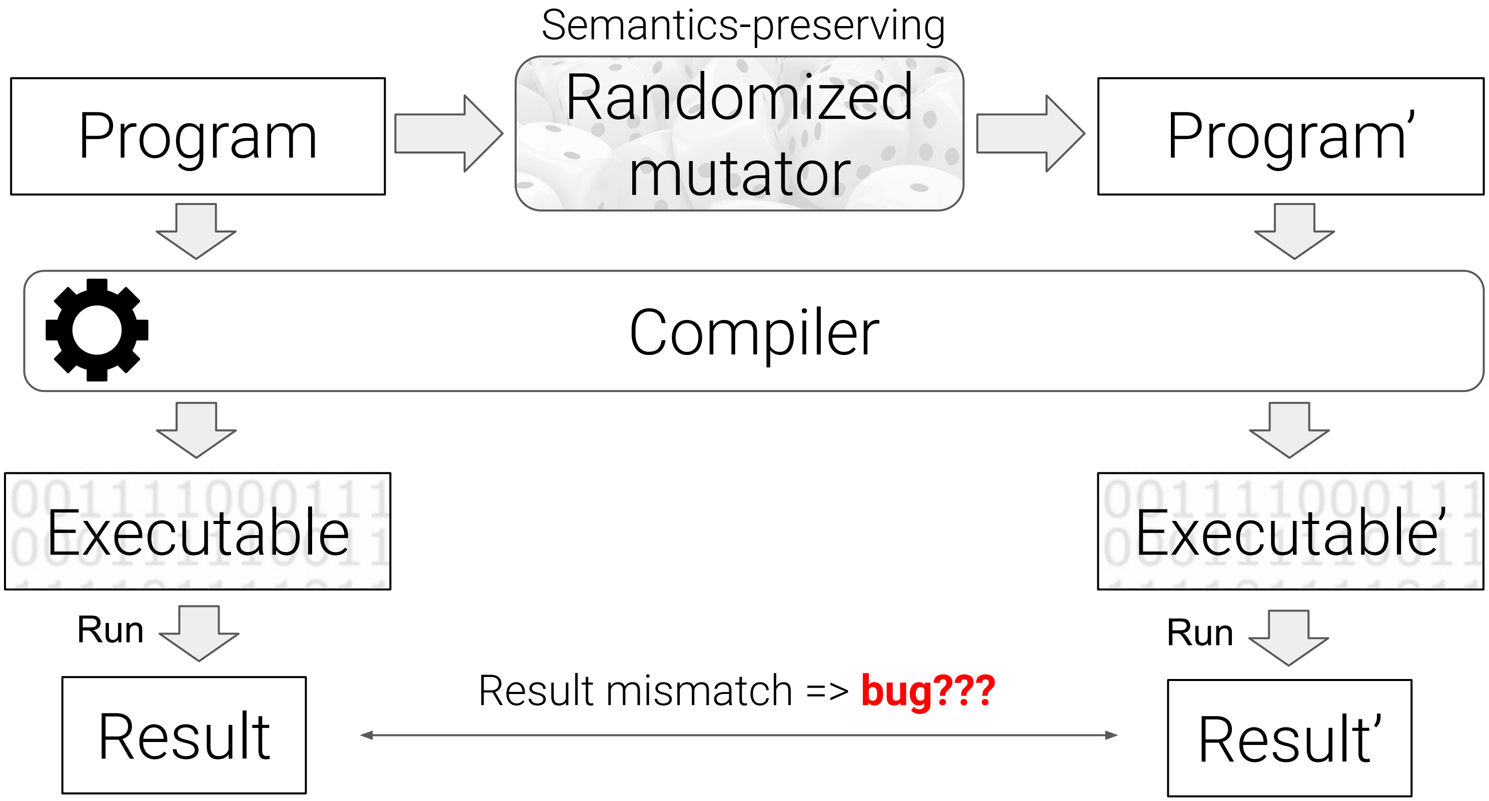
Result

Result'

Result mismatch => **bug!**

Le et. al, PLDI'14
Donaldson et al., OOPSLA'17

Pseudo-oracle: metamorphic testing



Success stories

Differential:

Yang et al., PLDI 2011
Most influential paper award at PLDI 2021

Success stories

Differential:

Yang et al., PLDI 2011
Most influential paper award at PLDI 2021

Metamorphic:

Equivalence Modulo Inputs Testing (EMI)

Le et al., PLDI 2014

Success stories

Differential:

Yang et al., PLDI 2011
Most influential paper award at PLDI 2021

Metamorphic:

Equivalence Modulo Inputs Testing (EMI)

Le et al., PLDI 2014

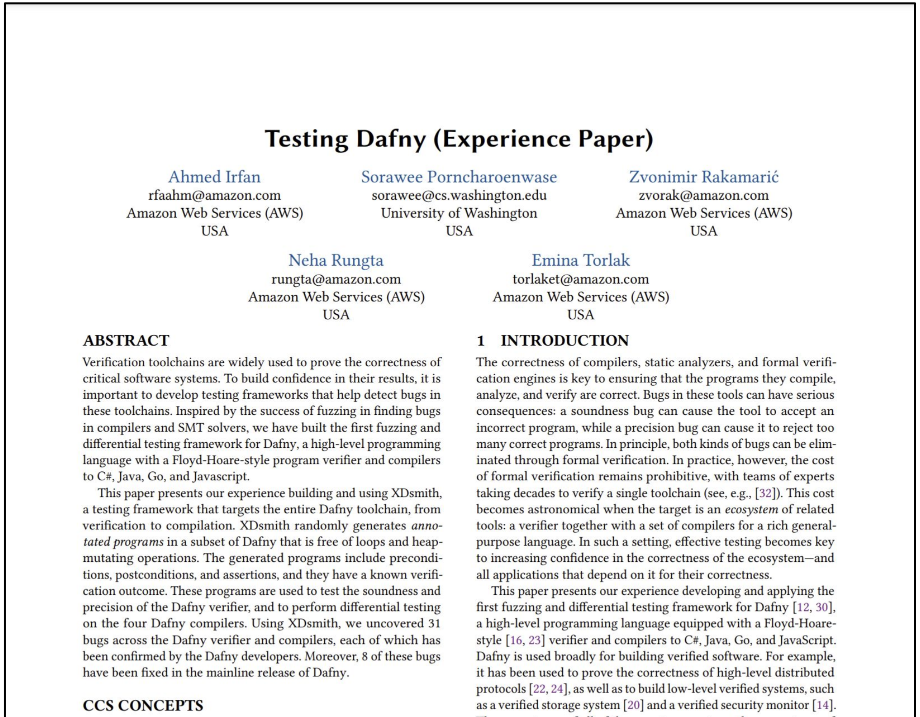
Led to finding and fixing of thousands of GCC and LLVM bugs

GraphicsFuzz: metamorphic testing for graphics compilers

Imperial College London → GraphicsFuzz → G

https://github.com/google/graphicsfuzz

Amazon: testing the Dafny verification language + compiler



Testing Dafny (Experience Paper)

Ahmed Irfan
rfaahm@amazon.com
Amazon Web Services (AWS)
USA

Sorawee Porncharoenwase
sorawee@cs.washington.edu
University of Washington
USA

Zvonimir Rakamarić
zvorak@amazon.com
Amazon Web Services (AWS)
USA

Neha Rungta
rungta@amazon.com
Amazon Web Services (AWS)
USA

Emina Torlak
torlaket@amazon.com
Amazon Web Services (AWS)
USA

ABSTRACT

Verification toolchains are widely used to prove the correctness of critical software systems. To build confidence in their results, it is important to develop testing frameworks that help detect bugs in these toolchains. Inspired by the success of fuzzing in finding bugs in compilers and SMT solvers, we have built the first fuzzing and differential testing framework for Dafny, a high-level programming language with a Floyd-Hoare-style program verifier and compilers to C#, Java, Go, and Javascript.

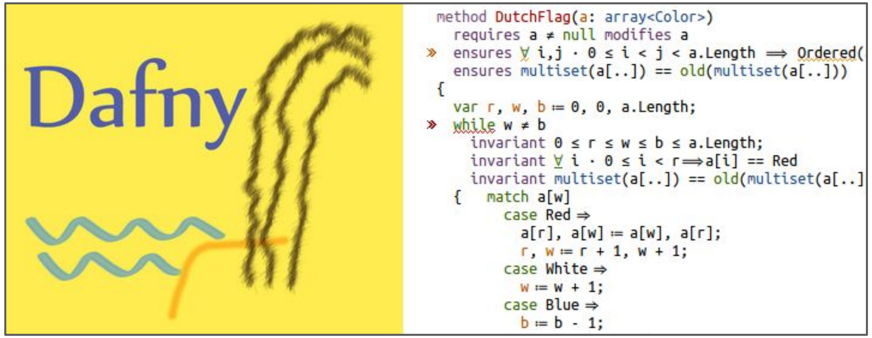
This paper presents our experience building and using XDsmith, a testing framework that targets the entire Dafny toolchain, from verification to compilation. XDsmith randomly generates *annotated programs* in a subset of Dafny that is free of loops and heap-mutating operations. The generated programs include preconditions, postconditions, and assertions, and they have a known verification outcome. These programs are used to test the soundness and precision of the Dafny verifier, and to perform differential testing on the four Dafny compilers. Using XDsmith, we uncovered 31 bugs across the Dafny verifier and compilers, each of which has been confirmed by the Dafny developers. Moreover, 8 of these bugs have been fixed in the mainline release of Dafny.

CCS CONCEPTS

1 INTRODUCTION

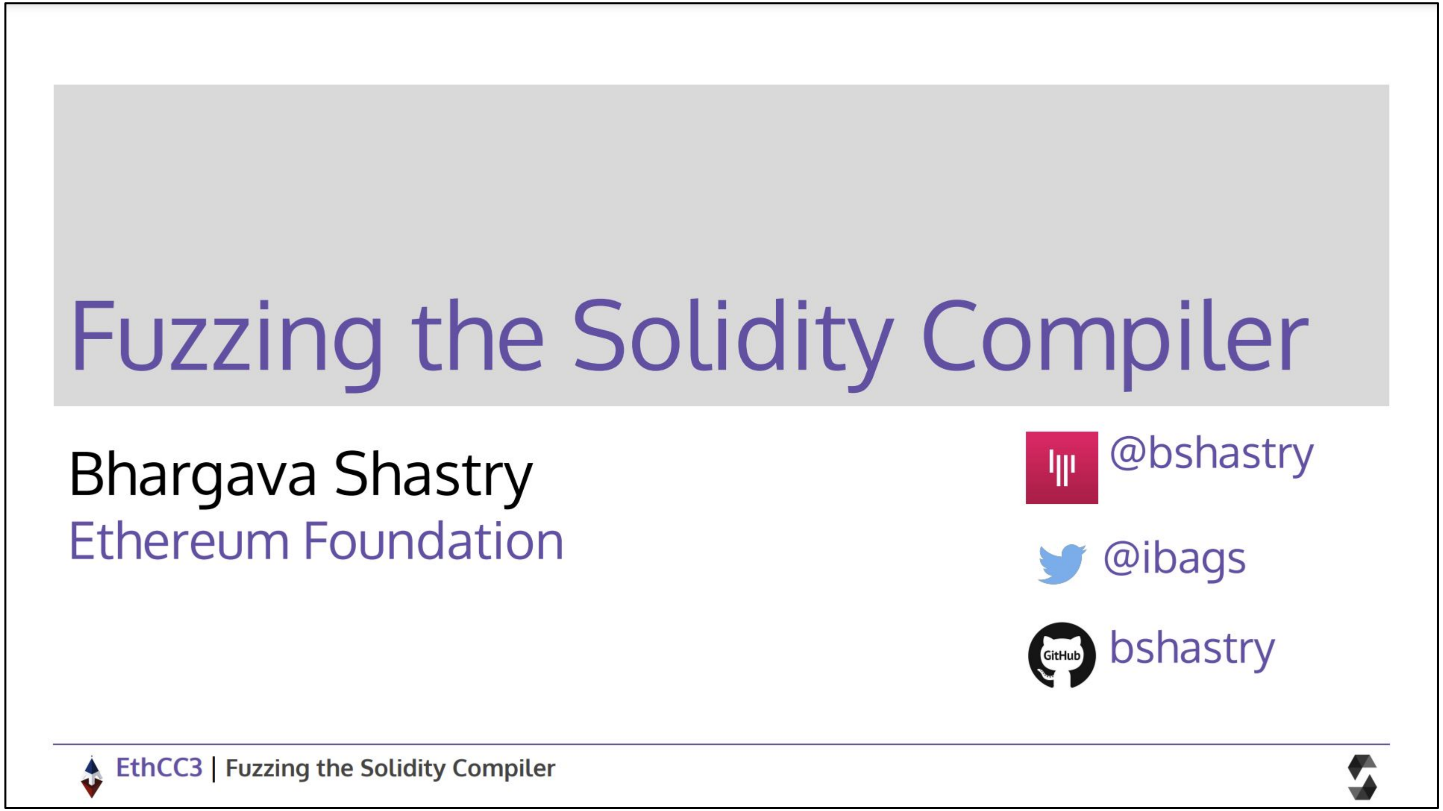
The correctness of compilers, static analyzers, and formal verification engines is key to ensuring that the programs they compile, analyze, and verify are correct. Bugs in these tools can have serious consequences: a soundness bug can cause the tool to accept an incorrect program, while a precision bug can cause it to reject too many correct programs. In principle, both kinds of bugs can be eliminated through formal verification. In practice, however, the cost of formal verification remains prohibitive, with teams of experts taking decades to verify a single toolchain (see, e.g., [32]). This cost becomes astronomical when the target is an *ecosystem* of related tools: a verifier together with a set of compilers for a rich general-purpose language. In such a setting, effective testing becomes key to increasing confidence in the correctness of the ecosystem—and all applications that depend on it for their correctness.

This paper presents our experience developing and applying the first fuzzing and differential testing framework for Dafny [12, 30], a high-level programming language equipped with a Floyd-Hoare-style [16, 23] verifier and compilers to C#, Java, Go, and JavaScript. Dafny is used broadly for building verified software. For example, it has been used to prove the correctness of high-level distributed protocols [22, 24], as well as to build low-level verified systems, such as a verified storage system [20] and a verified security monitor [14].



```
method DutchFlag(a: array<Color>)
  requires a ≠ null modifies a
» ensures ∀ i,j · 0 ≤ i < j < a.Length ⟹ Ordered(
  ensures multiset(a[..]) == old(multiset(a[..]))
{
  var r, w, b ≔ 0, 0, a.Length;
» while w ≠ b
    invariant 0 ≤ r ≤ w ≤ b ≤ a.Length;
    invariant ∀ i · 0 ≤ i < r⟹a[i] == Red
    invariant multiset(a[..]) == old(multiset(a[..]
  {   match a[w]
        case Red ⇒
          a[r], a[w] ≔ a[w], a[r];
          r, w ≔ r + 1, w + 1;
        case White ⇒
          w ≔ w + 1;
        case Blue ⇒
          b ≔ b - 1;
```

Solidity



Fuzzing the Solidity Compiler

Bhargava Shastry
Ethereum Foundation

@bshastry

@ibags

bshastry

EthCC3 | Fuzzing the Solidity Compiler

Writing randomized compiler testing tools isn’t that hard!

2021-2022 Imperial College Undergraduate projects:

- Hasan Mohsin: WebGPU shading language fuzzer
- Hana Watson: WebGPU shading language fuzzer
- Rayan Hatout: SPIR-V shading language fuzzer
- Mayank Sharma: Rust language fuzzer
- Kerry Xu: Rust language fuzzer

Talented students, but working alone and part time

Found dozens of bugs, achieved significant extra test coverage

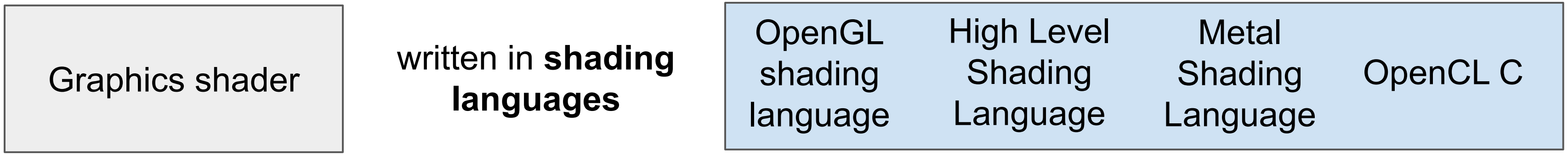
Part 2: Lightweight formal methods

Full blown compiler verification is largely out of scope

Major exception: **CompCert**

But: **major benefit** can be obtained by formalising **parts** of languages

Graphics shaders



Graphics shader

written in **shading languages**

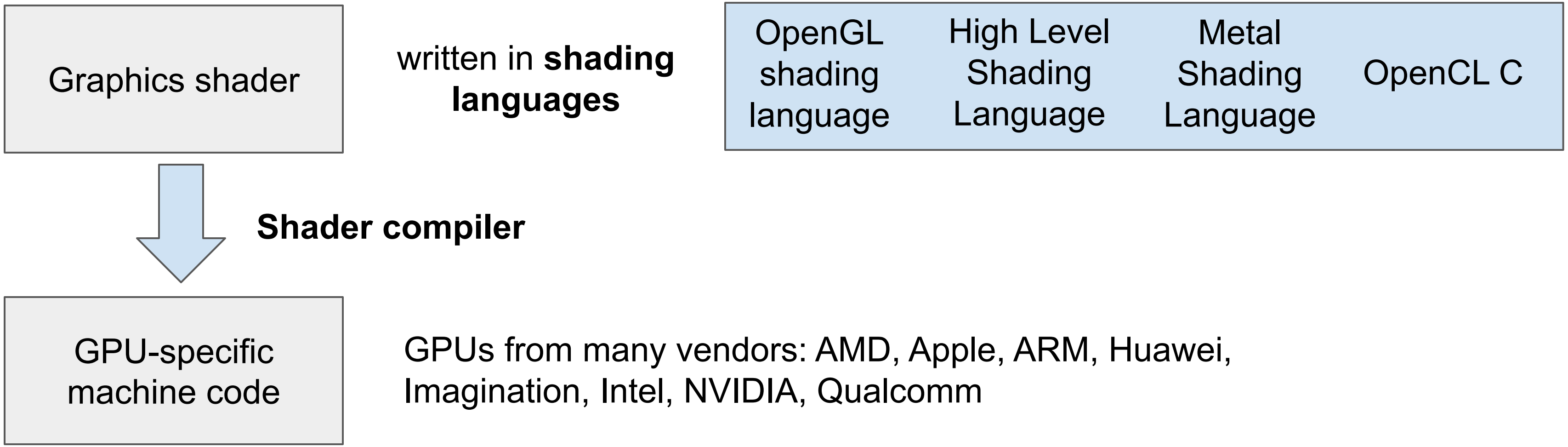
OpenGL shading language

High Level Shading Language

Metal Shading Language

OpenCL C

Graphics shaders



Graphics shader

written in **shading languages**

OpenGL shading language

High Level Shading Language

Metal Shading Language

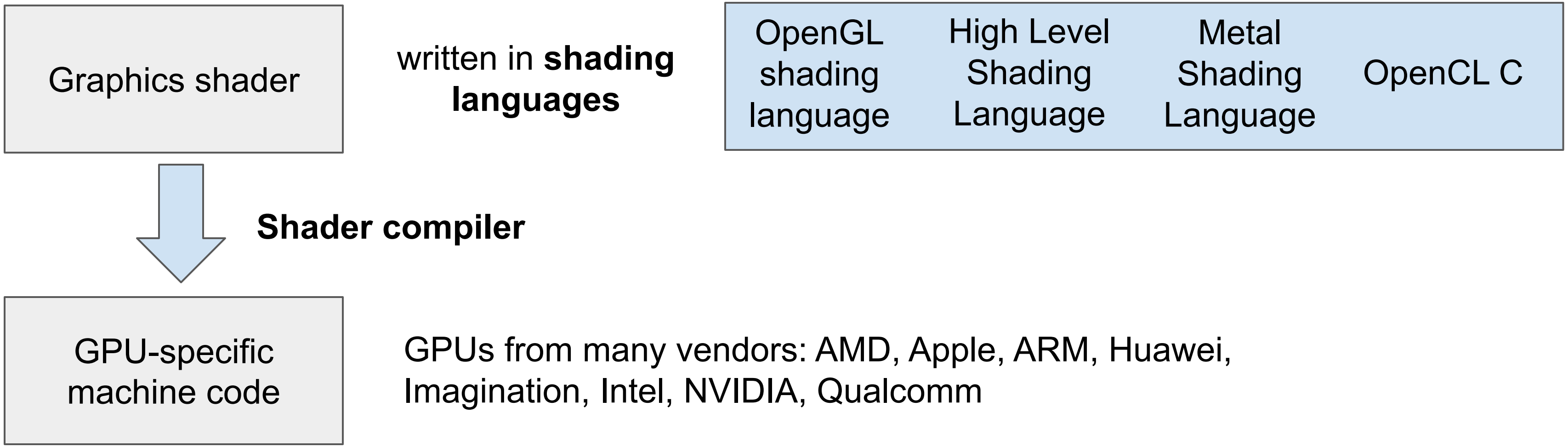
OpenCL C

Shader compiler

GPU-specific machine code

GPUs from many vendors: AMD, Apple, ARM, Huawei, Imagination, Intel, NVIDIA, Qualcomm

Graphics shaders



Graphics shader

written in **shading languages**

OpenGL shading language

High Level Shading Language

Metal Shading Language

OpenCL C

Shader compiler

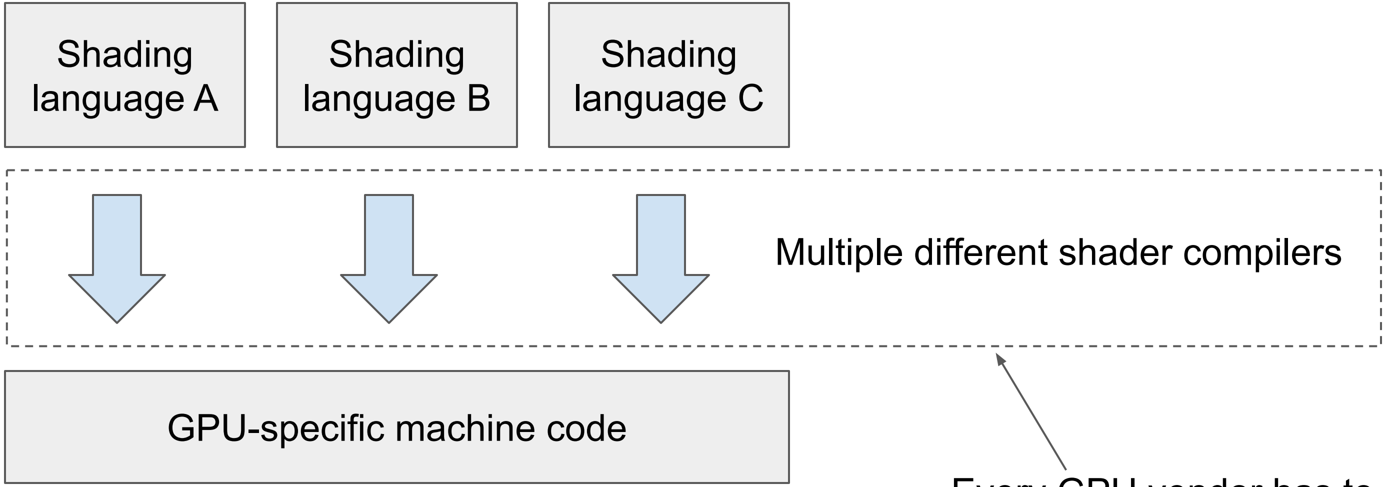
GPU-specific machine code

GPUs from many vendors: AMD, Apple, ARM, Huawei, Imagination, Intel, NVIDIA, Qualcomm

Shader compiler: the **most complex** part of a GPU device driver

SPIR-V: Standard, Portable Intermediate Representation

Motivation



Shading language A

Shading language B

Shading language C

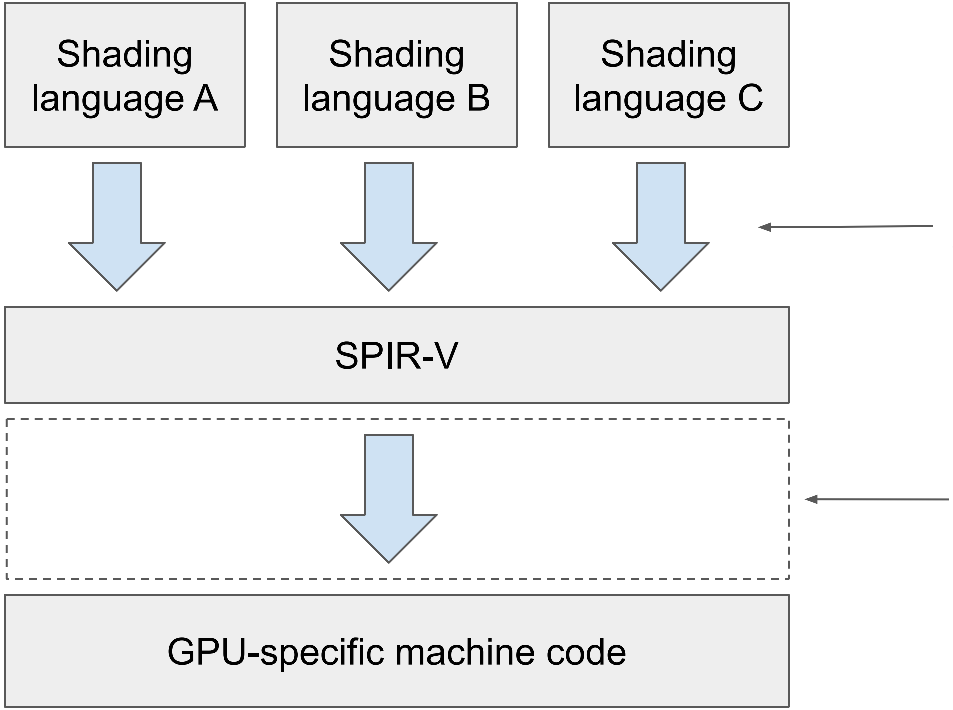
Multiple different shader compilers

GPU-specific machine code

Every GPU vendor has to maintain their own set of shader compilers: a lot of work

SPIR-V: Standard, Portable Intermediate Representation

Motivation



Shading language A

Shading language B

Shading language C

SPIR-V

GPU-specific machine code

Industry standard, GPU-agnostic translators

Every GPU vendor writes a compiler for SPIR-V - reduces overall burden

SPIR-V specification had some major problems

Problems related to sophisticated rules about control flow

Intended to help developers and compiler writers

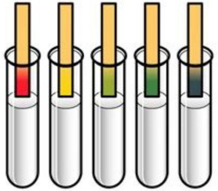
Not helping in practice:

- Dzmitry Malyshau, Mozilla: Horrors of SPIR-V
- Sean Baxter, Circle compiler: Targeting SPIR-V is super easy and the structurization requirements totally won't make you want to throw yourself off a cliff
- Hans-Kristian Arntzen, Arntzen Software: My personal hell of translating DXIL to SPIR-V

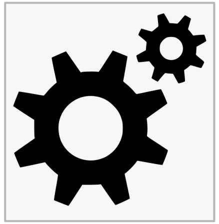
Sources of truth about SPIR-V

Prose specification

Conformance test suites



Validation tooling



Experts

David

Alan

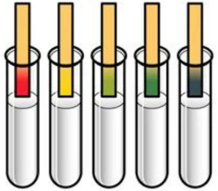
Modelling SPIR-V control flow in Alloy

Prose specification

Best-effort initial interpretation

Alloy model

Conformance test suites



Validation tooling

Experts

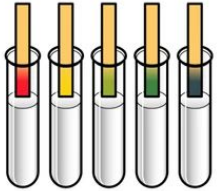
David

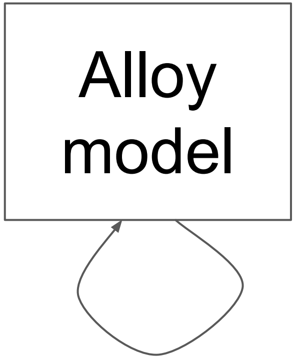
Alan

Modelling SPIR-V control flow in Alloy

Prose specification

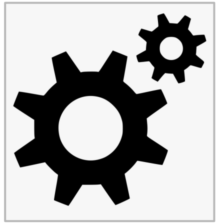
Conformance test suites





Formulate solutions to known problems

Validation tooling



Experts

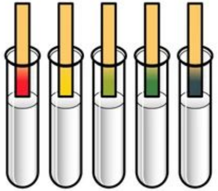
David

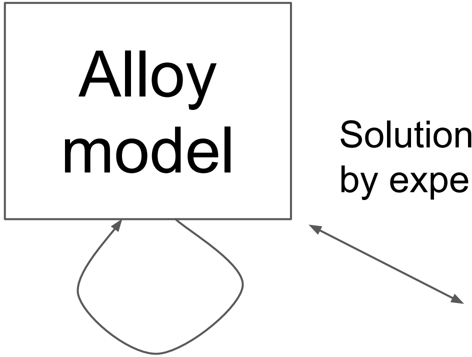
Alan

Modelling SPIR-V control flow in Alloy

Prose specification

Conformance test suites



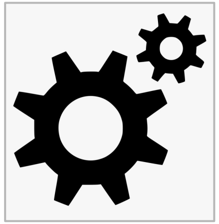


Alloy model

Solutions informed by experts

Formulate solutions to known problems

Validation tooling



Experts

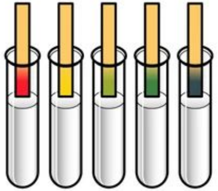
David

Alan

Modelling SPIR-V control flow in Alloy

Prose specification

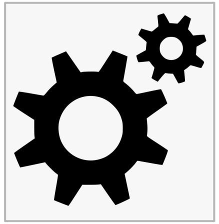
Conformance test suites



Cross-check
against test suites

Alloy
model

Validation tooling



Experts

David

Alan

Modelling SPIR-V control flow in Alloy

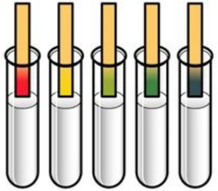
Prose specification

Validation tooling

Alloy model

Cross-check against test suites

Conformance test suites



Fix ill-formed tests

Consult with experts

Experts

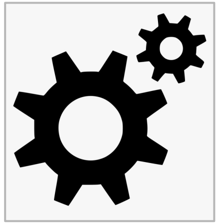
David

Alan

Modelling SPIR-V control flow in Alloy

Prose specification

Validation tooling

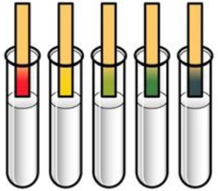


Alloy model

Fix flaws in model identified by tests

Cross-check against test suites

Conformance test suites



Fix ill-formed tests

Consult with experts

Experts

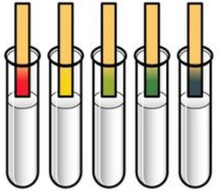
David

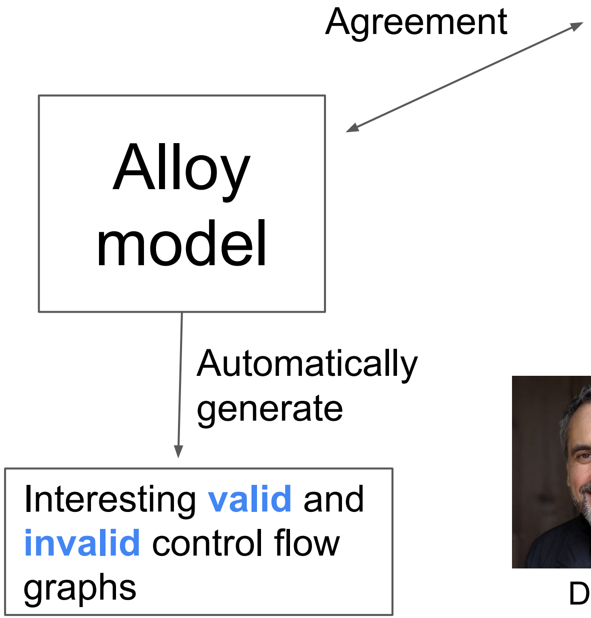
Alan

Modelling SPIR-V control flow in Alloy

Prose specification

Conformance test suites





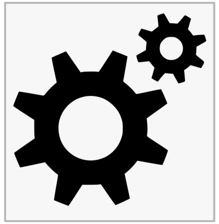
Agreement

Alloy model

Automatically generate

Interesting **valid** and **invalid** control flow graphs

Validation tooling

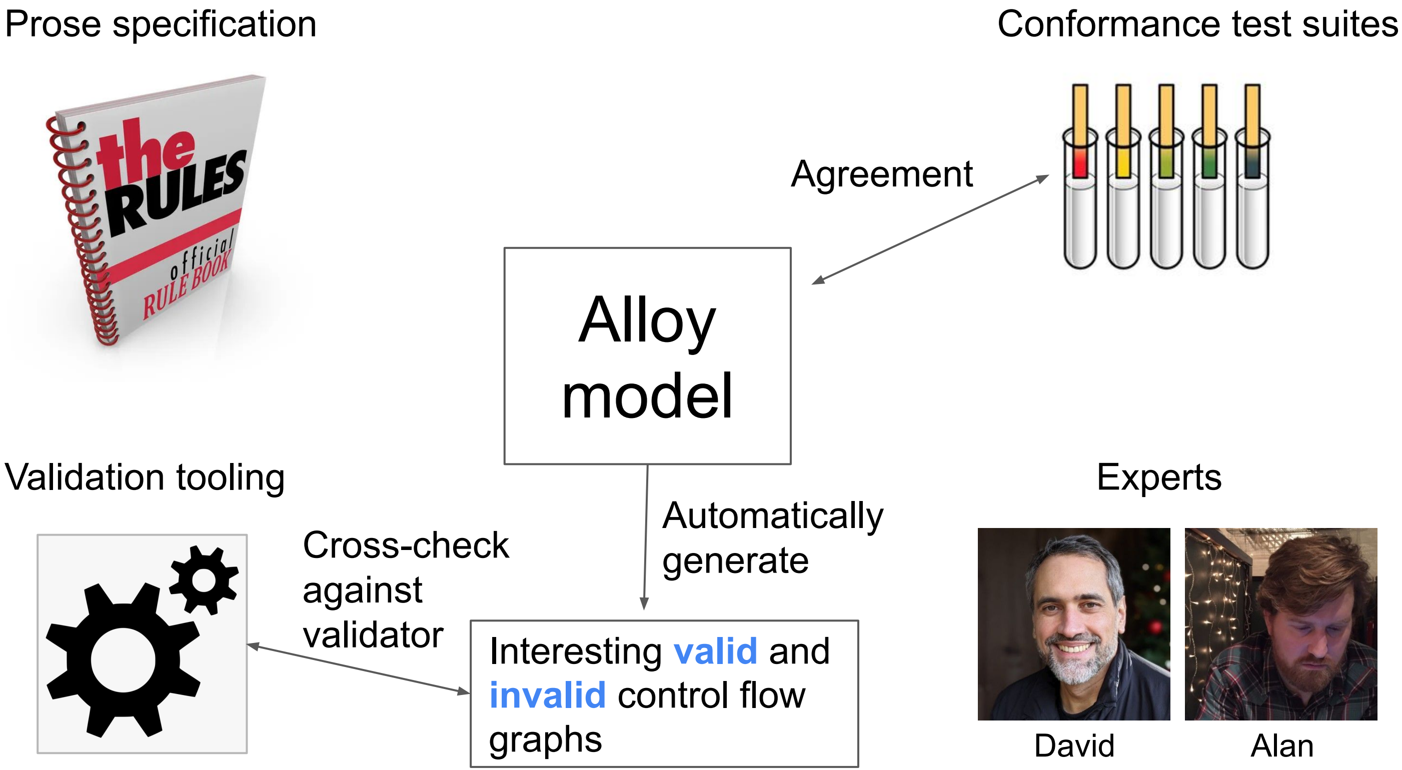


Experts

David

Alan

Modelling SPIR-V control flow in Alloy



Prose specification

Conformance test suites

Agreement

Alloy model

Validation tooling

Cross-check against validator

Automatically generate

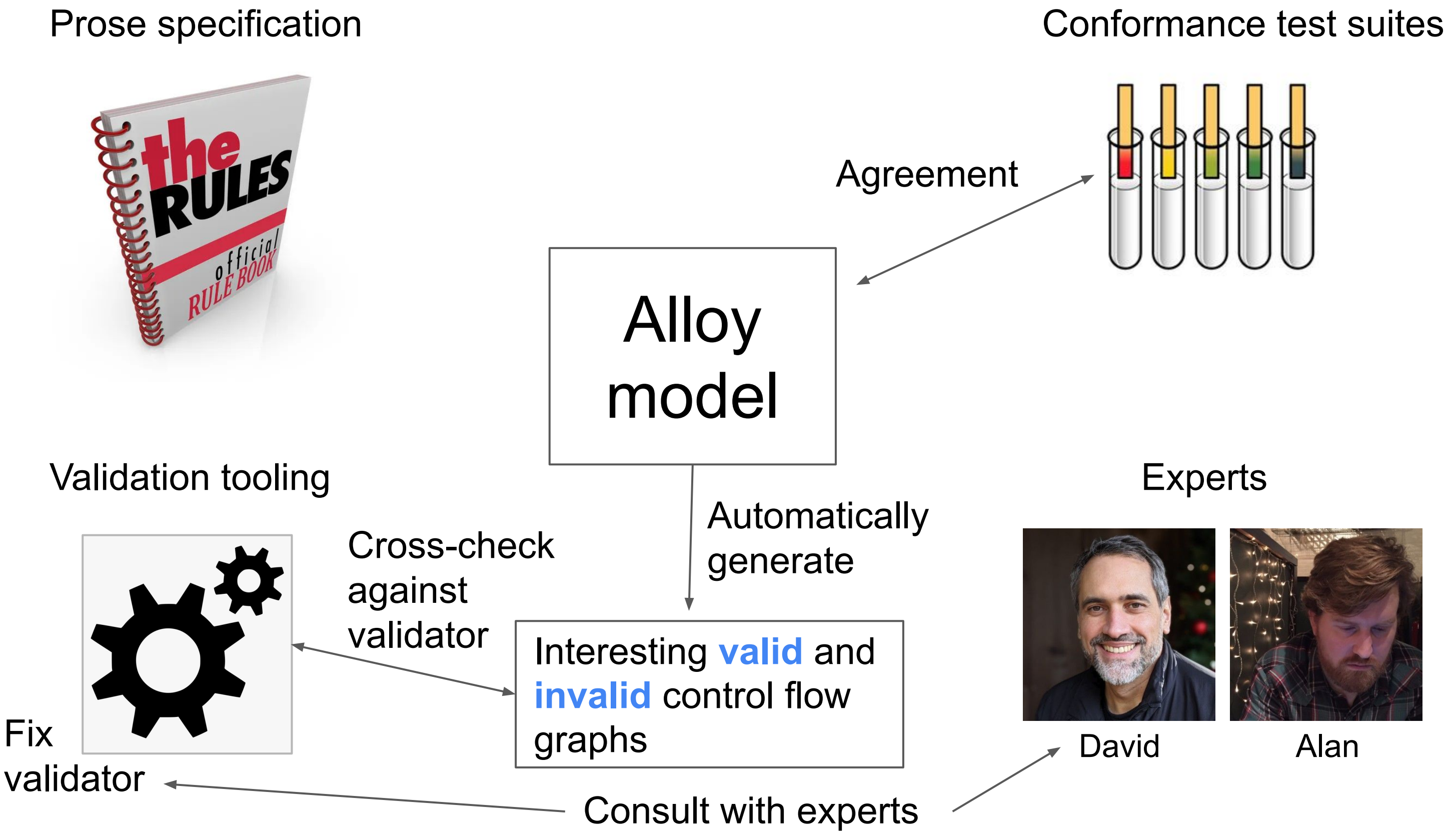
Interesting **valid** and **invalid** control flow graphs

Experts

David

Alan

Modelling SPIR-V control flow in Alloy



Prose specification

Conformance test suites

Agreement

Alloy model

Automatically generate

Interesting **valid** and **invalid** control flow graphs

Validation tooling

Cross-check against validator

Fix validator

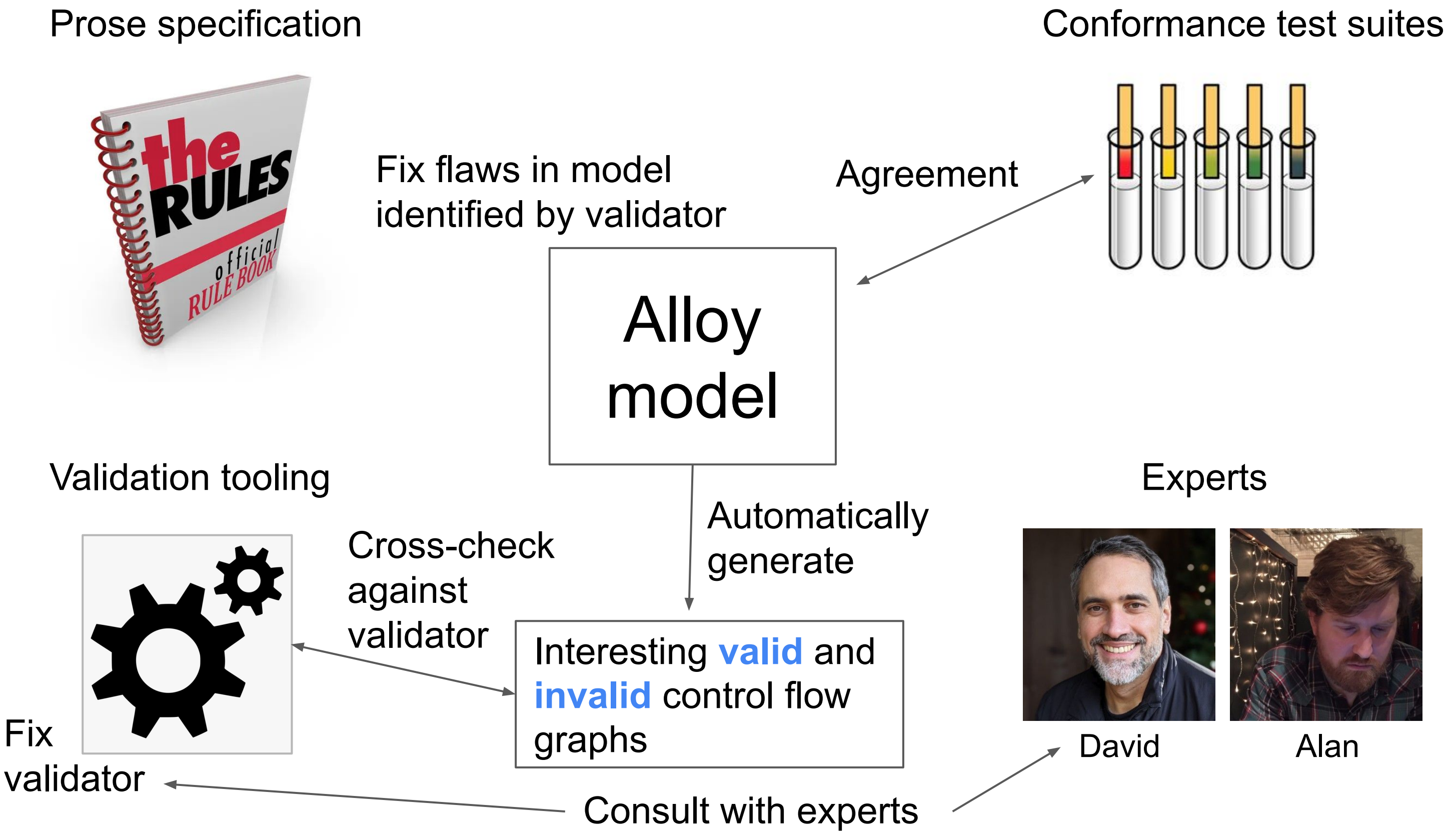
Consult with experts

Experts

David

Alan

Modelling SPIR-V control flow in Alloy



Prose specification

Conformance test suites

Fix flaws in model identified by validator

Agreement

Alloy model

Validation tooling

Automatically generate

Experts

Cross-check against validator

Interesting **valid** and **invalid** control flow graphs

David

Alan

Fix validator

Consult with experts

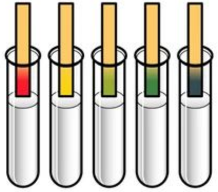
Modelling SPIR-V control flow in Alloy

Prose specification

Fix flaws in model identified by validator

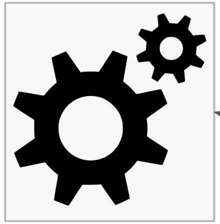
Cross-check against test suites

Conformance test suites



Alloy model

Validation tooling



Cross-check against validator

Automatically generate

Interesting **valid** and **invalid** control flow graphs

Experts

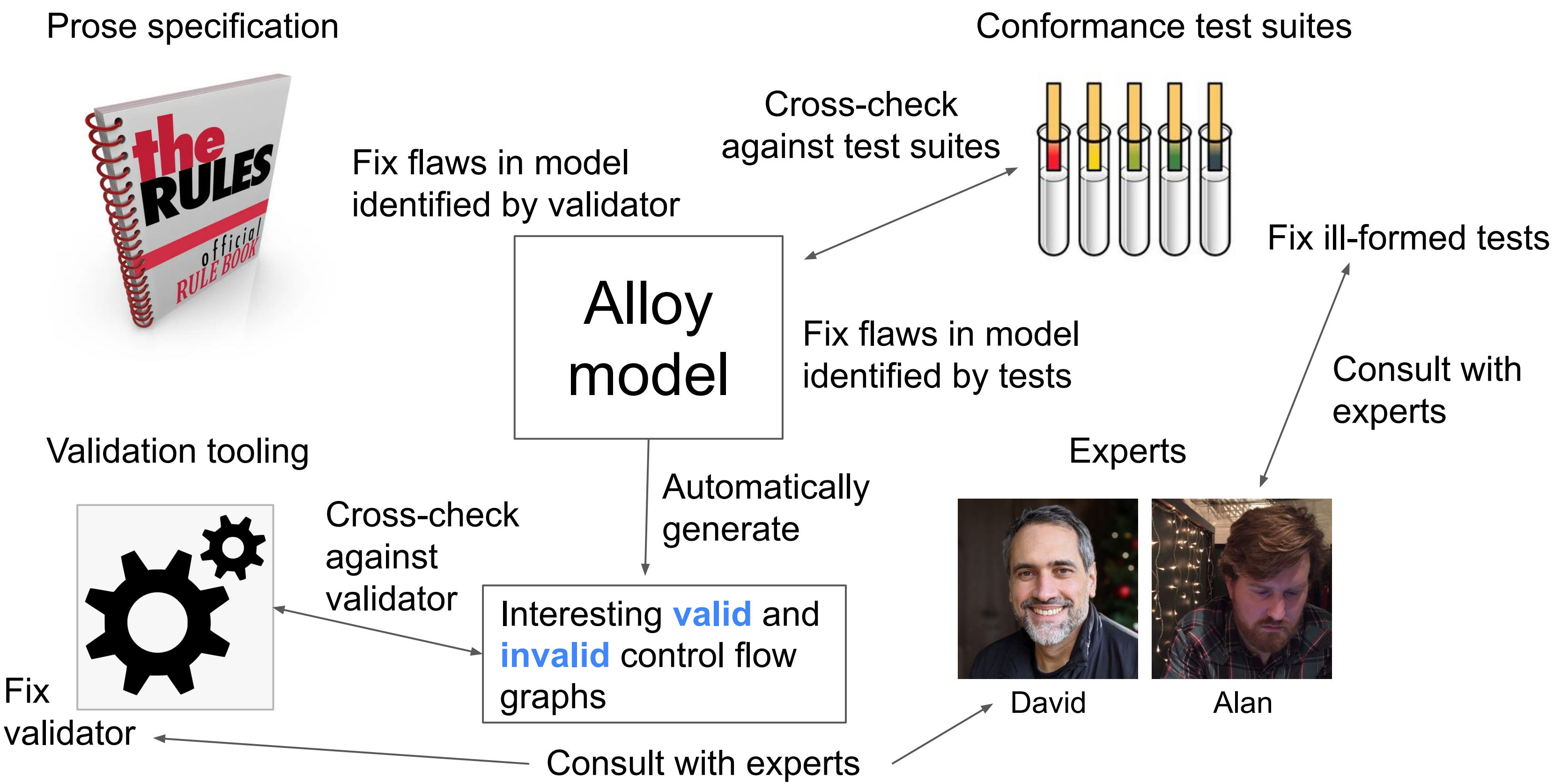
David

Alan

Fix validator

Consult with experts

Modelling SPIR-V control flow in Alloy



Prose specification

Fix flaws in model identified by validator

Cross-check against test suites

Conformance test suites

Alloy model

Fix flaws in model identified by tests

Fix ill-formed tests

Consult with experts

Validation tooling

Cross-check against validator

Automatically generate

Interesting **valid** and **invalid** control flow graphs

Experts

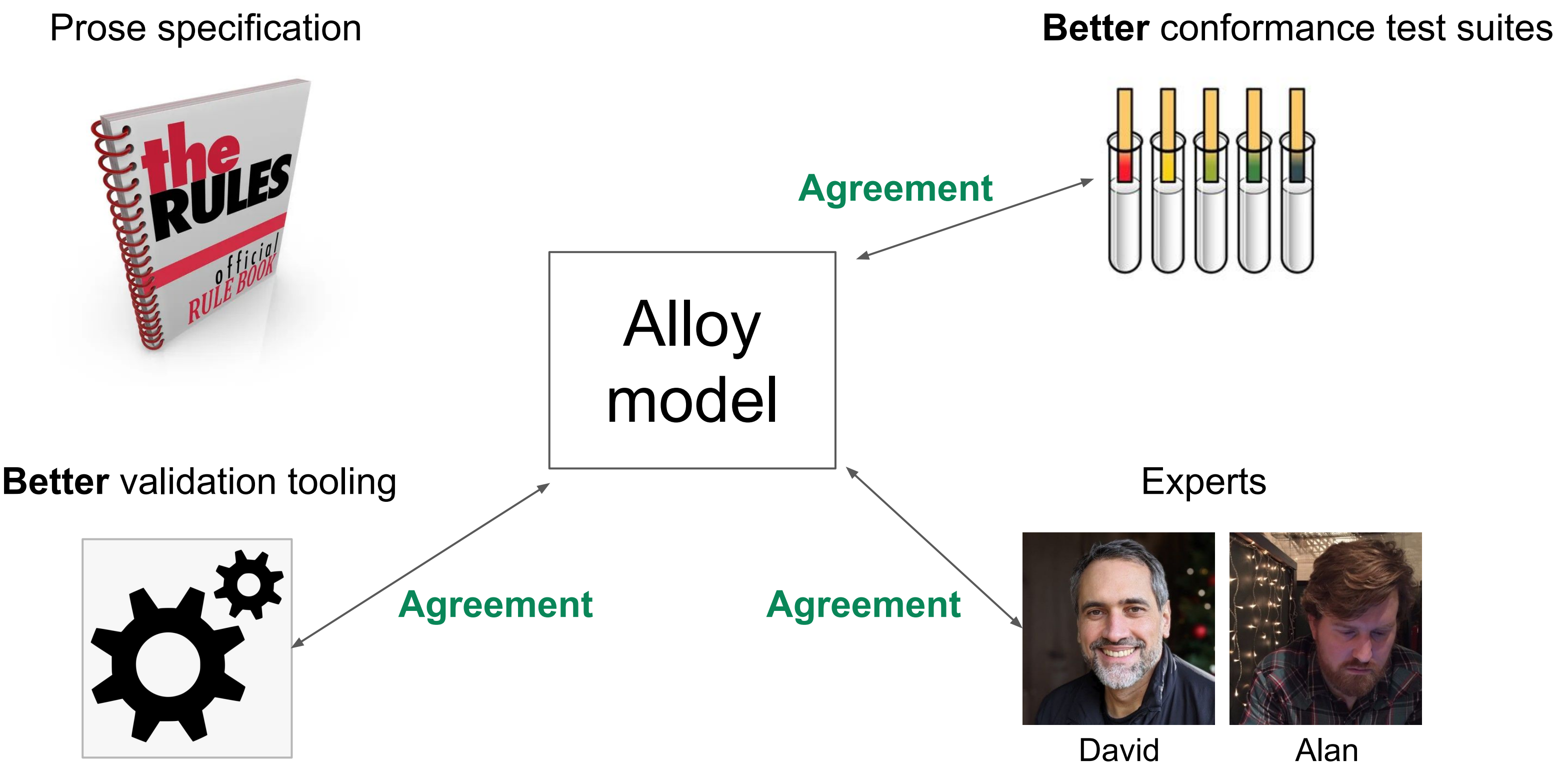
David

Alan

Fix validator

Consult with experts

Virtuous cycle improved formal model, conformance tests + tooling



Prose specification

Better conformance test suites

Agreement

Alloy model

Better validation tooling

Agreement

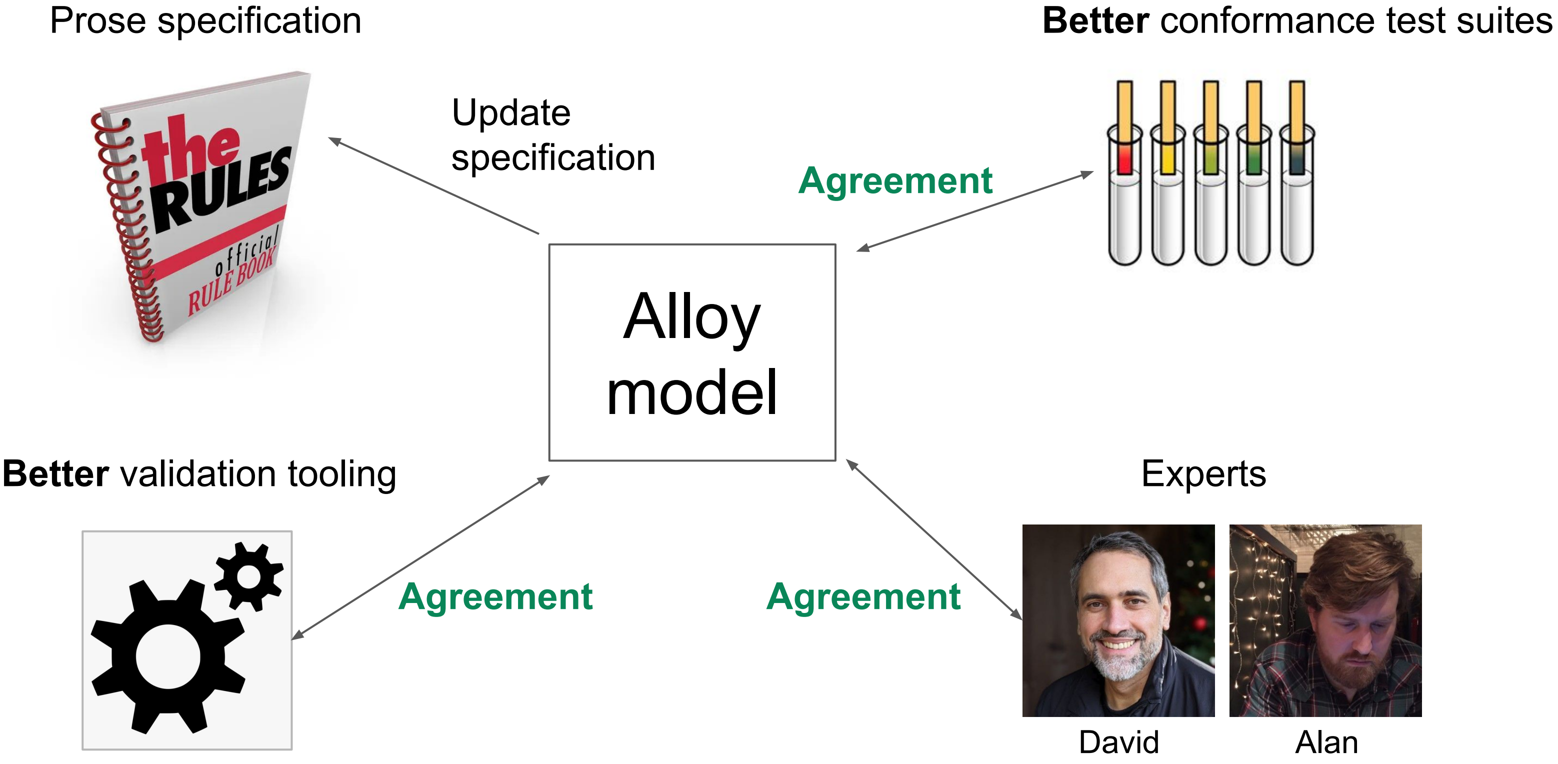
Agreement

Experts

David

Alan

Virtuous cycle improved formal model, conformance tests + tooling



Prose specification

Update specification

Agreement

Better conformance test suites

Alloy model

Better validation tooling

Agreement

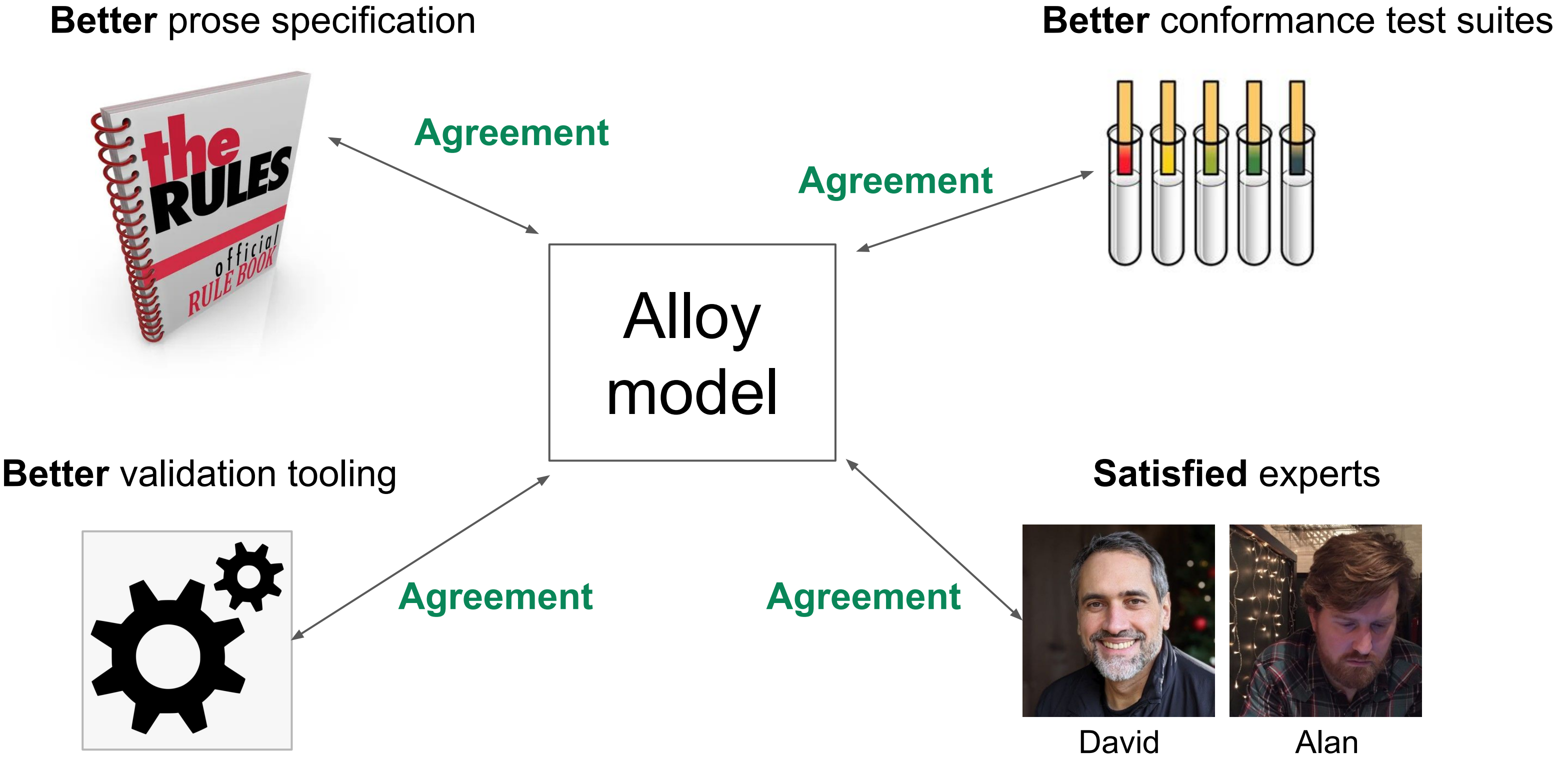
Agreement

Experts

David

Alan

Our changes are now integrated into the SPIR-V specification



Better prose specification

Agreement

Agreement

Better conformance test suites

Alloy model

Better validation tooling

Agreement

Agreement

Satisfied experts

David

Alan

Another lightweight formal methods success

Alive toolkit

Automatic verification of LLVM optimizations

Led to finding and fixing of many bugs

Formal guarantees for important LLVM peephole optimizations

Outlook

Randomized compiler testing is great

Lightweight formalization can be really useful

Can we:

- Combine them?
- Get a randomized tester automatically from a formal spec?
- Create a spectrum from lightweight to heavy-weight compiler validation?

afd@ic.ac.uk @afd_icl

Thank you!