Static analyzer debugging and quality assurance approaches

Maxim Menshikov

28.05.2020

St.Petersburg State University

About the author & the project

Maxim Menshikov —

- PhD student at St.Petersburg State University.
- Software engineer.
- (ex. security analyst; participated in commercial debugger project).

Equid[1] — a static analyzer for C/C++/RuC based on Model Checking and Abstract Interpretation. It verifies contracts and finds common defects.

[1] Engine for performing queries on unified intermediate representations of program and domain models

What's special in analyzers?

Not much.

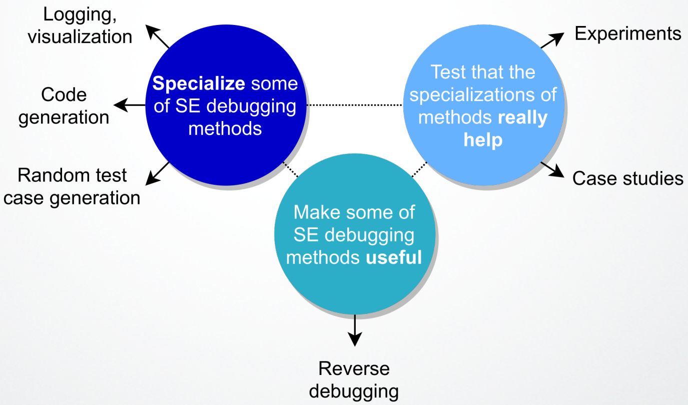
- Many equivalent transformations:
  input format $\neq$ intermediate format $\neq$ output format.
- Intermediate representations are mostly internal.
- The code is usually consistent and has high integrity, but there are logical mistakes, unprocessed parts $\rightarrow$ the biggest defects are logical.

The problem

- There are many debugging & quality assurance methods.
- None of them are specialized enough for static analysis.
- Every project brings its own set of hardly formalized methods.

What if we find a right specialization of the methods to the static analysis field?

The paper's goal



Defect sources (observations)

- Missing support for the specific syntax/intermediate representation (IR) construction in submodules.
- Small differences in implementations for repeating parts (classes).
- Transformation and ordering issues.

Defect reasons (observations)

- Low visuality of the transformation passes and the development process.
- Unattainable cross-dependencies between modules.
- Low quality of tests.

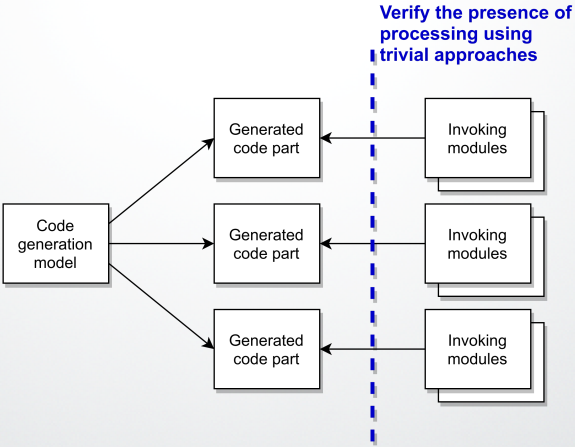
Proposed solutions

Proposing the solutions of these three groups:

- Code generation:
  Generated code usage verification.
- Testing:
  Goal-driven random test case generation.
- Logging:
  Log fusion and visual representation.

Code generation

- One model, several interpretations, many output source files.
- Perform a simple integrity check.



Code generation: enumeration example

```
model:
 - enum
 - enum_verify
 - command
 - vmir_mapping
data:
 cleantype: VmOpCode
 type: VmOpCode
 namespace: VirtualMachine
 command:
    - name: VmCommand_Base
      base:
        - ObjectWithProps
...
```

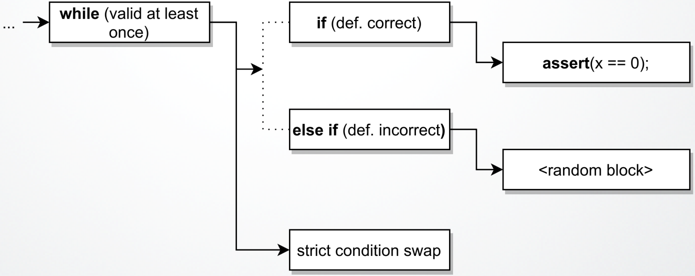
```
enum class VmOpCode
{
   Unknown,
   Declare,
   Init,
   NewItem,
   ...
};
```

```
std::string
    vmopcodeToToken(const VmOpCode &val);
...
```

```
core_indicate_use(VmOpCode,
  CoreEnumUse::AllVariants);
switch (command->getOpCode())
{
    case VmOpCode::Declare:
        ...
    /* Do we use all of enumeration items,
       and specifically NewItem? */
}
end_core_indicate_use(VmOpCode);
```

Goal-driven random test case generation

The idea is: generate input programs with an integrated verification goal (assertion).



Goal-driven random test case generation

1. The tool generates a random goal and asserts it $\rightarrow$ an expression.
2. The expression is repeatedly rolled into `if`/`switch`/`for`/`while`/... random blocks $\rightarrow$ a block.
3. The meaningful blocks are shuffled using equivalent transformations.

Goal-driven random test case generation

In result we get:

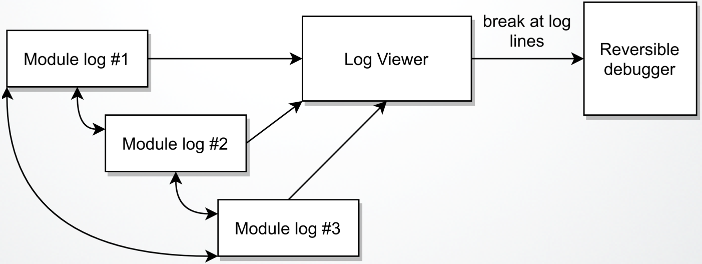
1. A completely random program.
2. A set of shuffled random programs.

By that, it is possible to verify:

1. Logical issues in transformations.
2. Ordering issues.
3. Runtime failures.

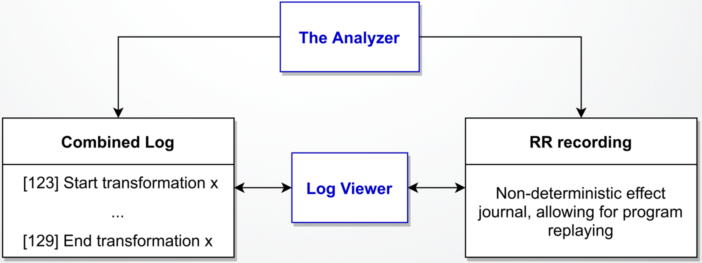
Log fusion

Fuse separate logs, set up cross-references, so the final log is a technical documentation of the run. Allows for easy navigation.



Log fusion: reverse recording assistance

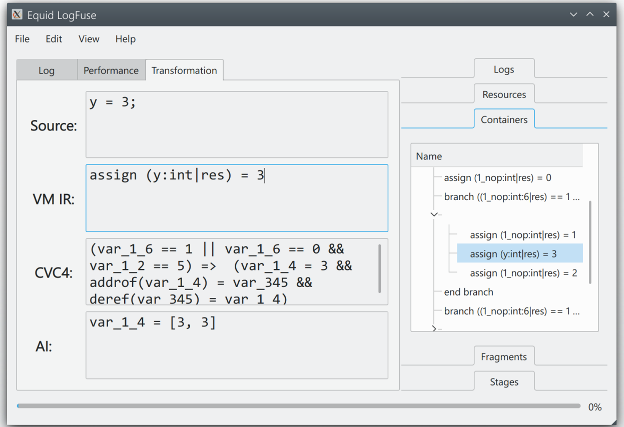
The Log Fusion also helps break right after the specific log line using reversible debugger like RR[2], UndoDB, etc. That is achieved using logging engine traps and GDB scripts.



[2] `https://rr-project.org`

Visual representation: steps

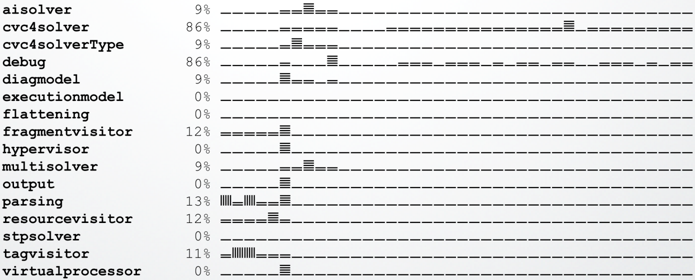
Visualize steps — present all transformations in one window, allow to debug specific transformations.



Visual representation: log health

Log health — visualize the time allocation for different modules.

In result, it is possible to determine whether the specific part is unintentially skipped.



Random test case generation: discovered issues & their severity

The method detected many performance, ordering, logical issues, and even runtime failures.

Defect type	Number of issues	Severity
Performance	3	Medium
Ordering	5	High
Runtime failure	1	High
Logical issues	1	Medium

Log fusion: (rough) time to resolve the issues

Average improvement rate: 2.8.

Defect type	Time to resolve before (h)	Time to resolve after (h)
Performance	25	13
Ordering	5	1
Runtime failure	1	0.3
Logical issues	1	1

Results: code generation and visual representation

The improvement is hard to examine.

In our experiment, developing the same feature twice took 7 times less time than on previous iteration - thanks to code generation.

Visual representation allowed to discover at least 2 performance issues, and overall provided an enormous help during defect resolution.

Conclusion

- The specialization of the proposed methods helps find real issues in the static analyzer.
- The combination of approaches dramatically decreases the defect resolution time.