An Efficient Runtime Validation Framework based on the Theory of Refinement

Mitesh Jain, Synopsys
Pete Manolios, Northeastern University
Property-based Testing Methodology

Given an implementation

1. Define a set of properties

2. Design a test suite and define oracles

3. Execute tests and check for property violations

4. Analyze the violations and fix
   - Implementation
   - Properties
Property-based Testing Methodology

Given an implementation

1. Define a set of properties

2. Design a test suite and define oracles

3. Execute tests and check for property violations

4. Analyze the violations and fix
   - Implementation
   - Properties

Difficult to determine if the specification is complete
Property-based Testing Methodology

Given an implementation

1. Define a set of properties

2. Design a test suite and define oracles

3. Execute tests and check for property violations

4. Analyze the violations and fix
   - Implementation
   - Properties

Difficult to determine if the specification is complete

Defining oracles is expensive and error-prone
Property-based Testing Methodology

Given an implementation

1. Define a set of properties
2. Design a test suite and define oracles
3. Execute tests and check for property violations
4. Analyze the violations and fix
   - Implementation
   - Properties

Difficult to determine if the specification is complete
Defining oracles is expensive and error-prone
Changes in specification
Property-based Testing Methodology

Implementation

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>RF</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
</tbody>
</table>

Set of Properties

1. Correct arithmetic operations
2. Detecting and stalling the pipeline on data hazards
3. Branch/Jump instructions
4. ...
Property-based Testing Methodology

Implementation

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>RF</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
</tbody>
</table>

Set of Properties

1. Correct arithmetic operations
2. Detecting and stalling the pipeline on data hazards
3. Branch/Jump instructions
4. ...
Trends in Functional Verification: A 2014 Industry Study

Figure 17. Root Cause of Functional Flaws

1Harry Foster, DAC 2015 (reproduced)
Trends in Functional Verification: A 2014 Industry Study\textsuperscript{1}

Figure 17. Root Cause of Functional Flaws

Refinement-based testing methodology

\textsuperscript{1}Harry Foster, DAC 2015 (reproduced)
Trends in Functional Verification: A 2014 Industry Study

Refinement-based testing methodology

Compile the refinement conjecture to a runtime check

Figure 17. Root Cause of Functional Flaws

1Harry Foster, DAC 2015 (reproduced)
Refinement-based Testing Methodology

Implementation

Concrete System

Abstract System

Specification

Instruction Set Architecture
- add rd, ra, rb
- sub rd, ra, rb
- jnz imm
- . . .
Refinement-based Testing Methodology

### Implementation

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>RF</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>RF</td>
<td>EX</td>
<td>WB</td>
</tr>
</tbody>
</table>

### Specification

Instruction Set Architecture

- `add rd, ra, rb`
- `sub rd, ra, rb`
- `jnz imm`
- ...
Challenges

- $\mathcal{A}$ and $\mathcal{C}$ may differ
  - Data representation
  - Number of state components
  - Atomicity of a computation step
  - ...

Relate behaviors of systems expressed at different levels of abstraction
Notions Of Refinement

- Bisimulation
- Simulation
- Stuttering Bisimulation
- Stuttering Simulation
- Skipping Simulation

Strongest

Weakest
Applications

- Microprocessor verification
- Compiler verification
- Distributed/Concurrent systems
- Microkernels
- ...
Applications

- Microprocessor verification
- Compiler verification
- Distributed/Concurrent systems
- Microkernels
- ...

Local characterizations for effective reasoning
Refinement Conjecture
A local characterization of Skipping Simulation

one step
\( \langle \exists v: w \rightarrow v: uBv \rangle \)

skipping on the right
\( \langle \exists v: w \rightarrow v: uBv \rangle \)

stuttering on left
\((uBw \land rankT(u, w) \prec rankT(s, w))\)

stuttering on right
\(\langle \exists v: w \rightarrow v: sBv \land \)
\( rankL(v, s, u) \prec rankL(w, s, u) \rangle \)
Refinement Conjecture
A local characterization of Skipping Simulation

\[ \langle \exists v: w \rightarrow v: uBv \rangle \]

Concrete system does not skip

\[ \langle \exists v: w \rightarrow v: sBv \wedge \text{stuttering on right} \rangle \]

\[ \langle \exists v: w \rightarrow v: uBv \wedge \text{stuttering on left} \rangle \]
Refinement Conjecture
A local characterization of Skipping Simulation

Concrete system does not skip

Abstract system does not stutter

\[ S \rightarrow^1 W \]
\[ U \rightarrow^1 V \]

one step
\[ \langle \exists v: w \rightarrow v: uBv \rangle \]

stuttering on left
\[ (uBw \land rankT(u, w) < rankT(s, w)) \]
Algorithm 1: Refinement Check

\textbf{Input} : \(s\): concrete system state
\hspace{1em} \(n\): number of steps to run
\hspace{1em} \(r\): refinement map
\hspace{1em} \(\text{rank}_T\): rank of concrete state

\textbf{Output}: Partition, Error Status

\(w \leftarrow r(s); \text{error} \leftarrow \text{false}; \text{partition} \leftarrow \langle \rangle; i \leftarrow 0; j \leftarrow 0;\)

\textbf{do}

\hspace{1em} \(u \leftarrow \text{Select-concrete-next-state}(s);\)

\hspace{2em} \langle \text{match}, v \rangle \leftarrow \text{Match-abstract-next-state}(w, u);\)

\hspace{2em} \textbf{if} match

\hspace{3em} \text{partition} \leftarrow \text{partition} :: \langle i, j \rangle;

\hspace{3em} i \leftarrow i + 1; j \leftarrow j + 1;

\hspace{3em} s \leftarrow u; w \leftarrow v;\)

\hspace{2em} \textbf{else if}

\hspace{3em} \(r(u) = w \land \text{rank}_T(u, w) < \text{rank}_T(s, w)\)

\hspace{4em} i \leftarrow i + 1;

\hspace{4em} s \leftarrow u;

\hspace{2em} \textbf{else error} \leftarrow \text{true};\)

\hspace{1em} \(n \leftarrow n - 1;\)

\textbf{while} \(n > 0 \land \neg\text{error};\)

\textbf{return} \langle \text{partition}, \text{error} \rangle;
Refinement Conjecture
Runtime Checker

\[ \neg \exists v : w \rightarrow v : uBv \]

\[ \text{Compile} \]

\[ \langle \text{partition}, \text{error} \rangle \]

Local proof method $\rightsquigarrow$ efficient runtime refinement checker
Evaluation: RISC-V Sodor

- 5-stage pipeline Sodor Processor
  - Single issue in-order pipeline processor
  - Supports full bypassing between functions units
- Spike, executable RISC-V ISA simulator
  - High-level specification
  - Serves as the oracle
Evaluation

- Effectiveness in detecting bugs
- Overhead cost of the refinement checker
Evaluation

- Effectiveness in detecting bugs

<table>
<thead>
<tr>
<th>Errors Injected</th>
<th>Detect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction classification</td>
<td>✓</td>
</tr>
<tr>
<td>Control: Stall Mechanism</td>
<td>✓</td>
</tr>
<tr>
<td>Control: Pipeline hazard detection</td>
<td>✓</td>
</tr>
<tr>
<td>Arithmetic-logic unit</td>
<td>✓</td>
</tr>
<tr>
<td>Load store unit</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Overhead cost of the refinement checkers
Evaluation

- Effectiveness in detecting bugs

<table>
<thead>
<tr>
<th>Errors Injected</th>
<th>Detect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction classification</td>
<td>✓</td>
</tr>
<tr>
<td>Control: Stall Mechanism</td>
<td>✓</td>
</tr>
<tr>
<td>Control: Pipeline hazard detection</td>
<td>✓</td>
</tr>
<tr>
<td>Arithmetic-logic unit</td>
<td>✓</td>
</tr>
<tr>
<td>Load store unit</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Overhead cost of the refinement checkers
Evaluation

- Effectiveness in detecting bugs
- Overhead cost of the refinement checker

1. Match concrete and abstract states
2. Computing the refinement map
3. Computing the rank of a concrete state

**Algorithm 1: Refinement Check**

```plaintext
... 
do
    u ← Select-concrete-next-state(s);
    ⟨match, v⟩ ← Match-abstract-next-state(w, u);

    if match then
        partition ← partition :: ⟨i, j⟩;
        i ← i + 1; j ← j + 1;
        s ← u; w ← v;
    end

    else if r(u) = w ∧ rankT(u) < rankT(s)
    then
        i ← i + 1;
        s ← u;
    end

    else error ← true;

    n ← n − 1;

while n > 0 ∧ ¬error;
return ⟨partition, error⟩;
```
Evaluation

- Effectiveness in detecting bugs
- Overhead cost of the refinement checker

1. Match concrete and abstract states
2. Computing the refinement map
3. Computing the rank of a concrete state
Conclusion

An alternate testing methodology based on the theory of refinement.

Refinement implies functional correctness

Algorithm 1: Refinement Check

<table>
<thead>
<tr>
<th>Input</th>
<th>s: concrete system state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n: number of steps to run</td>
</tr>
<tr>
<td></td>
<td>r: refinement map</td>
</tr>
<tr>
<td></td>
<td>rankT: rank of concrete state</td>
</tr>
</tbody>
</table>

| Output | Partition, Error Status |

w ← r(s); error ← false; partition ← ∅; i ← 0; j ← 0;

do

u ← Select-concrete-next-state(s);

⟨match, v⟩ ← Match-abstract-next-state(w, u);

if match

    partition ← partition :: ⟨i, j⟩;
    i ← i + 1; j ← j + 1;
    s ← u; w ← v;

else if

    r(u) = w ∧ rankT(u, w) < rankT(s, w)

        i ← i + 1;
        s ← u;

    else error ← true;

n ← n − 1;

while n > 0 ∧ ¬error;

return ⟨partition, error⟩;
Conclusion

An alternate testing methodology based on the theory of refinement.

Refinement implies functional correctness

Abstract system serves as the oracle

Algorithm 1: Refinement Check

Input: : s: concrete system state
n: number of steps to run
r: refinement map
rankT: rank of concrete state

Output: Partition, Error Status
w ← r(s); error ← false; partition ← ⟨⟩; i ← 0;
  j ← 0;
  do
    u ← Select-concrete-next-state(s);
    ⟨match, v⟩ ← Match-abstract-next-state(w, u);
    if match
      partition ← partition :: ⟨i, j⟩;
      i ← i + 1 ; j ← j + 1;
      s ← u ; w ← v;
    else if
      r(u) = w ∧ rankT(u, w) < rankT(s, w)
      i ← i + 1;
      s ← u;
    else error ← true;
    n ← n − 1;
  while n > 0 ∧ ¬error;
return ⟨partition, error⟩;
Conclusion

An alternate testing methodology based on the theory of refinement.

Refinement implies functional correctness

Abstract system serves as the oracle

Robust to changes in the implementation

Algorithm 1: Refinement Check

Input:
- $s$: concrete system state
- $n$: number of steps to run
- $r$: refinement map
- $\text{rank}T$: rank of concrete state

Output:
- Partition, Error Status

$w \leftarrow r(s); \text{error} \leftarrow \text{false}; \text{partition} \leftarrow \langle \rangle; i \leftarrow 0;\ j \leftarrow 0$

do

$u \leftarrow \text{Select-concrete-next-state}(s)$

$\langle \text{match}, v \rangle \leftarrow \text{Match-abstract-next-state}(w, u)$

if match

$\text{partition} \leftarrow \text{partition} :: \langle i, j \rangle$

$i \leftarrow i + 1; \ j \leftarrow j + 1$

$s \leftarrow u; \ w \leftarrow v$

else if $r(u) = w \land \text{rank}T(u, w) \prec \text{rank}T(s, w)$

$i \leftarrow i + 1$

$s \leftarrow u$

else error $\leftarrow \text{true}$

$n \leftarrow n - 1$

while $n > 0 \land \neg\text{error}$

return $\langle \text{partition}, \text{error} \rangle$. 

Thank You
References

- An efficient refinement-based testing methodology

**Skipping refinement**
Mitesh Jain and Pete Manolios
CAV, 2015, 2017

**Proving skipping refinement using ACL2s**
Mitesh Jain and Pete Manolios
ACL2, 2015