The Yogi Project
Software property checking via verification and testing

Aditya V. Nori, Sriram K. Rajamani
Programming Languages and Tools
Microsoft Research India
What is Yogi?

- An industrial strength program verifier
- Philosophy: Synergize verification and testing
- **Synergy [FSE ’06]**, **Dash [ISSTA ‘08]**, **Smash [POPL ‘10]**, **Bolt [submitted]** algorithms to perform scalable analysis
- Engineered a number of optimizations for scalability
- Integrated with Microsoft’s Static Driver Verifier (SDV) toolkit and used internally
Property checking

```c
void f(int *p, int *q)
{
  0:   *p = 4;
  1:   *q = 5;
  2:   assert (¬ϕ_error)
}
```

Question
Does the assertion hold for all possible inputs?

- **Must analysis:** finds bugs, but can’t prove their absence
- **May analysis:** can prove the absence of bugs, but can result in false errors

More generally, we are interested in the query

$$\langle ϕ_{pre} \Rightarrow f \phi_{error}\rangle$$
Must information

\[ \langle T \Rightarrow_f (* p \not= 4) \rangle = yes \]

```
void f(int *p, int *q)
{
    0:   *p = 4;
    1:   *q = 5;
}
```

- Captures facts that are guaranteed to hold on particular executions of the program (*under-approximation*)
- Error condition is reachable by any input that satisfies \((p = q)\)
May information

\[(p \neq q) \Rightarrow f(* p \neq 4) = no\]

void f(int *p, int&q)
{
    0:  *p = 4;
    1:  *q = 5;
}

- Captures facts that are true for all executions of the program (over-approximation)
- Proof can be obtained by keeping track of the predicates \((p = q)\) and \((* p \neq 4)\)
Dash: Proofs from Tests

- Algorithm uses only test case generation operations
- Maintains two data structures:
  - A forest of reachable concrete states (tests)
    - Under-approximates executions of the program
  - A region graph (an abstraction)
    - Over-approximates all executions of the program
- Our goal: bug finding and proving
  - If a test reaches an error, we have found a bug
  - If we refine the abstraction so that there is *no* path from the initial region to error region, we have a proof
- Key ideas
  - Frontier
  - $WP_\alpha$ uses only aliases $\alpha$ that are present along concrete tests that are executed
Step 1: Try to generate a test that crosses the frontier

- Perform symbolic simulation on the path until the frontier and generate a constraint $\varphi_1$
- Conjoin with the condition $\varphi_2$ needed to cross frontier
- Is $\varphi_1 \land \varphi_2$ satisfiable?
Key ideas

Step 1: Try to generate a test that crosses the frontier
- Perform symbolic simulation on the path until the frontier and generate a constraint $\varphi_1$
- Conjoin with the condition $\varphi_2$ needed to cross frontier
- Is $\varphi_1 \land \varphi_2$ satisfiable? [YES]

Step 2: run the test and extend the frontier
Key ideas

Step 1: Try to generate a test that crosses the frontier
- Perform symbolic simulation on the path until the frontier and generate a constraint $\varphi_1$
- Conjoin with the condition $\varphi_2$ needed to cross frontier
- Is $\varphi_1 \land \varphi_2$ satisfiable? [NO]

Step 2: use $WP_\alpha$ to refine so that the frontier moves back!
Can extend test beyond frontier?

Refine abstraction

Construct initial abstraction
Construct random tests

Test succeeded?

Abstraction succeeded?

\( \tau = \) error path in abstraction
\( f = \) frontier of error path

Can extend test beyond frontier?

\( \tau = (0, 1, 2, 3, 4, 7, 8, 9) \)

Symbolic execution + Theorem proving

Input:
Program \( P \)
Property \( \psi \)

void \( f(\text{int} \ y) \)
{
  0: \ int \ lock, x;
  1: \ do \{
  2: \ \ \ lock = 1;
  3: \ \ \ x = y;
  4: \ \ \ if (*) \{
  5: \ \ \ \ lock = 0;
  6: \ \ \ \ y = y+1;
  7: \ \ \} \ while (x != y)
  8: \ \ \ if (lock != 1)
  9: \ \ \ \ error();
 10: \ \}

Refinement

\[ \rho = (lock.state \neq L) \]
Refinement

\[
p = (\text{lock.state} \neq L)
\]
Another iteration

Input:
Program $P$
Property $\psi$

Construct initial abstraction
Construct random tests

Test succeeded?
yes
Bug!
no

Abstraction succeeded?
yes
Proof!
no

$\tau =$ error path in abstraction
$f =$ frontier of error path

Can extend test beyond frontier?
yes
no

Refine abstraction

void $f(int\ y)$
{
  0: int lock, x;
  1: do {
  2:    lock = 1;
  3:    x = y;
  4:    if (*) {
  5:      lock = 0;
  6:      y = y+1;
  7: } while (x != y)
  8:     if (lock != 1)
  9:     error();
 10: }

$\tau = (0,1,2,3,4,7,9)$
Correct, the program is ...

```c
void f(int y)
{
    int lock, x;
    do {
        lock = 1;
        x = y;
        if (*) {
            lock = 0;
            y = y+1;
        }
    } while (x != y)
    if (lock != 1)
        error();
}
```
Key idea
Perform a recursive Dash query on the called procedure and use the result to either generate a test or compute $WP_\alpha$
Interprocedural analysis

Dash(\varphi_1 \Rightarrow_{foo} \varphi_2)
- pass: perform refinement
- fail: generate test
Procedure summaries

- A **must summary** for a procedure $\mathcal{P}_i$ is of the form $(\varphi_1, \varphi_2) \in \mathop{\text{must}}\limits_{\mathcal{P}_i}$
- $\forall t \in \varphi_2 . \exists s \in \varphi_1 . t$ can be obtained by executing $\mathcal{P}_i$ from an initial state $s$

- A $\neg$**may summary** for a procedure $\mathcal{P}_i$ is of the form $(\varphi_1, \varphi_2) \in \mathop{\text{\neg may}}\limits_{\mathcal{P}_i}$
- $\forall s \in \varphi_1 \forall t \in \varphi_2 . t$ cannot be obtained by executing $\mathcal{P}_i$ starting in state $s$
Compositional may-must analysis

\[ \varphi_1 \in \Pi_{n_1}, \varphi_2 \in \Pi_{n_2}, \varphi_1 \cap \Omega_{n_1} \neq \emptyset, \varphi_2 \cap \Omega_{n_2} = \emptyset \]

\[ e = (n_1, n_2) \in E_{P_i} \text{ is a call to procedure } P_j \]

\[ (\hat{\varphi}_1, \hat{\varphi}_2) \in \text{must}_{P_j} \]

\[ \Omega_{n_1} \supseteq \hat{\varphi}_1, \theta \subseteq \hat{\varphi}_2, \varphi_2 \cap \theta \neq \emptyset \]

\[ \Omega_{n_2} := \Omega_{n_2} \cup \theta \]

\[ \text{[MUST – POST – USESUM]} \]

**must summary**

- \( \hat{\varphi}_1 \subseteq \Omega_{n_1} \)
- \( P_j \)
- \( (\hat{\varphi}_2 \supseteq \theta) \land (\varphi_2 \cap \theta \neq \emptyset) \)

**procedure** \( P_i \)

- Check if frontier \( (n_1, n_2) \) can be extended by a **must summary** \( (\hat{\varphi}_1, \hat{\varphi}_2) \)
- If yes, grow \( \Omega_{n_2} \) with \( \theta \subseteq \hat{\varphi}_2 \)
Compositional may-must analysis

\[ \phi_1 \in \Pi_{n_1} \quad \phi_2 \in \Pi_{n_2} \quad \phi_1 \cap \Omega_{n_1} \neq \emptyset \quad \phi_2 \cap \Omega_{n_2} = \emptyset \]

\[ e = (n_1, n_2) \in E_{P_i} \text{ is a call to procedure } P_j \]

\[ (\phi_1, \phi_2) \xrightarrow{\text{must}}_{P_j} \Omega_{n_1} \supseteq \hat{\phi}_1 \quad \theta \subseteq \hat{\phi}_2 \quad \phi_2 \cap \theta \neq \emptyset \]

\[ \Omega_{n_2} := \Omega_{n_2} \cup \theta \]

[MUST – POST – USESUM]

- Check if frontier \((n_1, n_2)\) can be extended by a must summary \((\hat{\phi}_1, \hat{\phi}_2)\)
- If yes, grow \(\Omega_{n_2}\) with \(\theta \subseteq \hat{\phi}_2\)
Compositional may-must analysis

\[ \varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad \varphi_1 \cap \Omega_{n_1} \neq \emptyset \quad \varphi_2 \cap \Omega_{n_2} = \emptyset \]

\[ e = (n_1, n_2) \in E_{P_i} \text{ is a call to procedure } P_j \]

\[ \langle \hat{\varphi}_1, \hat{\varphi}_2 \rangle \in \overset{\neg \text{may}}{\longrightarrow}_{P_j} \quad \varphi_2 \subseteq \hat{\varphi}_2 \quad \theta \subseteq \hat{\varphi}_1 \quad \neg \theta \cap \Omega_{n_1} = \emptyset \]

\[ \Pi_{n_1} := (\Pi_{n_1} \setminus \{\varphi_1\}) \cup \{\varphi_1 \cap \theta, \varphi_1 \cap \neg \theta\} \]

\[ N_e := N_e \cup \{(\varphi_1 \cap \theta, \varphi_2)\} \]

- Check if frontier \((n_1, n_2)\) can be refined by a \(\neg\text{may summary} \ (\hat{\varphi}_1, \hat{\varphi}_2)\)
- If yes, use \(\theta \subseteq \hat{\varphi}_1\) to refine the abstraction
- If both \textit{must} and \(\neg\text{may} \) summaries are not available, analyze procedure \(P_j\)
  - \textit{yes} \(\Rightarrow \text{must summary} \) for \(P_j\)
  - \textit{no} \(\Rightarrow \neg\text{may summary} \) for \(P_j\)
Optimizations

- Engineering for making Yogi robust, scalable and industrial strength

- Several of the implemented optimizations are folklore
  - Very difficult to design tools that are bug free \(\Rightarrow\) evaluating optimizations is hard!
  - Our empirical evaluation gives tool builders information about what gains can be realistically expected from optimizations
  - Details in ICSE ‘10

- Vanilla implementation of algorithms:
  - (flpydisk, cancelSpinLock) took 2 hours

- Algorithms + engineering + optimizations:
  - (flpydisk, cancelSpinLock) took less than 1 second!
Benchmarks:
- 30 WDM drivers and 83 properties (2490 runs)
- Anecdotal belief: most bugs in the tools are usually caught with this test suite
### Empirical results (Summaries)

<table>
<thead>
<tr>
<th>Summaries</th>
<th>Total time (minutes)</th>
<th>#defects</th>
<th>#timeouts</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>2160</td>
<td>241</td>
<td>77</td>
</tr>
<tr>
<td>no</td>
<td>3780</td>
<td>236</td>
<td>165</td>
</tr>
</tbody>
</table>

42%
Bolt: a generic framework that uses MapReduce style parallelism to scale top-down analysis.
Example

```c
int foo(int p_foo);
int bar(int p_bar);
int baz(int p_baz);

main(int i, int j){
    int x, y;
    if (i > 0)
        x = foo(i);
    else if (j > -10)
        x = bar(i);
    else
        x = baz(j);
    y = x + 5;
    assert(y > 0);
}
```
Empirical results

~Linear speedup!

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time taken (sequential)</td>
<td>26 hours</td>
</tr>
<tr>
<td>Total time taken (parallel)</td>
<td>7 hours</td>
</tr>
<tr>
<td>Average observed speedup</td>
<td>3.71x</td>
</tr>
<tr>
<td>Maximum observed speedup</td>
<td>7.41x</td>
</tr>
</tbody>
</table>
Questions?

PLDI 2012 tutorial