# DynaMoth: Dynamic Code Synthesis for Automatic Program Repair

AST 2016

Thomas Durieux & Martin Monperrus

March 6, 2017

Inria & University of Lille

# Automatic test-suite based repair

DynaMoth is an automatic test-suite based repair tools.

**Automatic test-suite based repair**

Generate a patch according to the test-suite.

# Repair domain

## Buggy condition

```
-   if (u * v == 0) {
+   if (u == 0 || v == 0) {
      return (Math.abs(u) + Math.abs(v));
    }
```

## Missing pre-condition

```
+ if (specific != null) {
    sb.append(": "); // sb is a StringBuilder
+ }
```

# DynaMoth

**Problem**

Too simple patch

**Solution**

Improve patch synthesis by supporting method call in the patches

Example:

```
@@ -258,2 +258,4 @@ class Complex implements FieldElement,
    Serializable  {
  if (divisor.isZero) {
+   if (0 == this.multiply(this).getReal())
      return NaN;
+   }
```

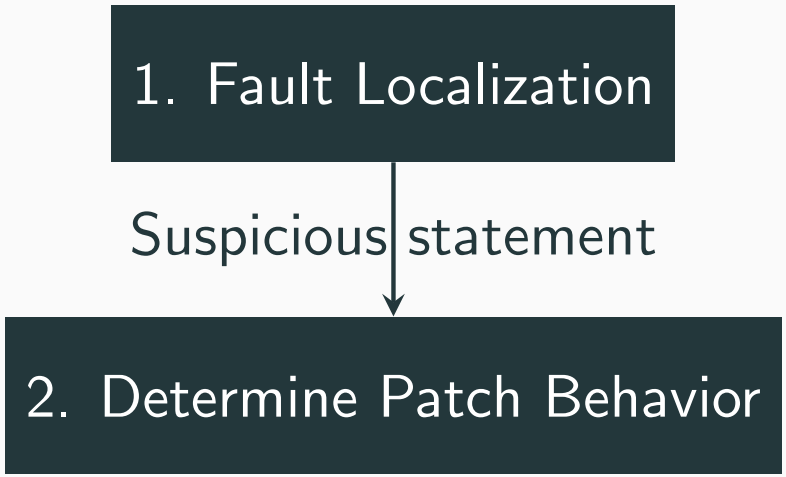
# DynaMoth

The three main steps:

1. Fault Localization

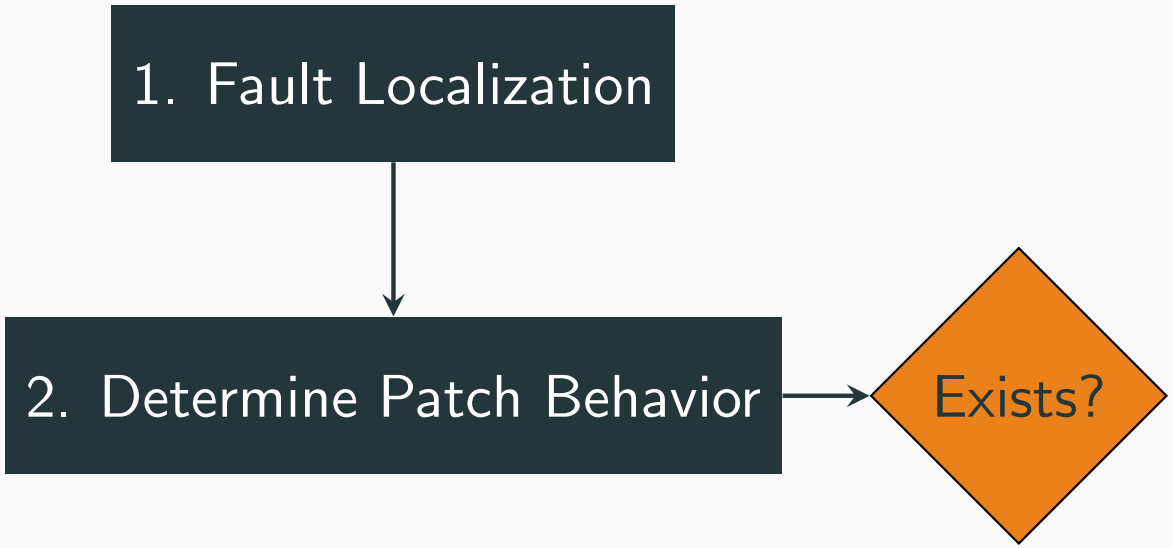
# DynaMoth

The three main steps:



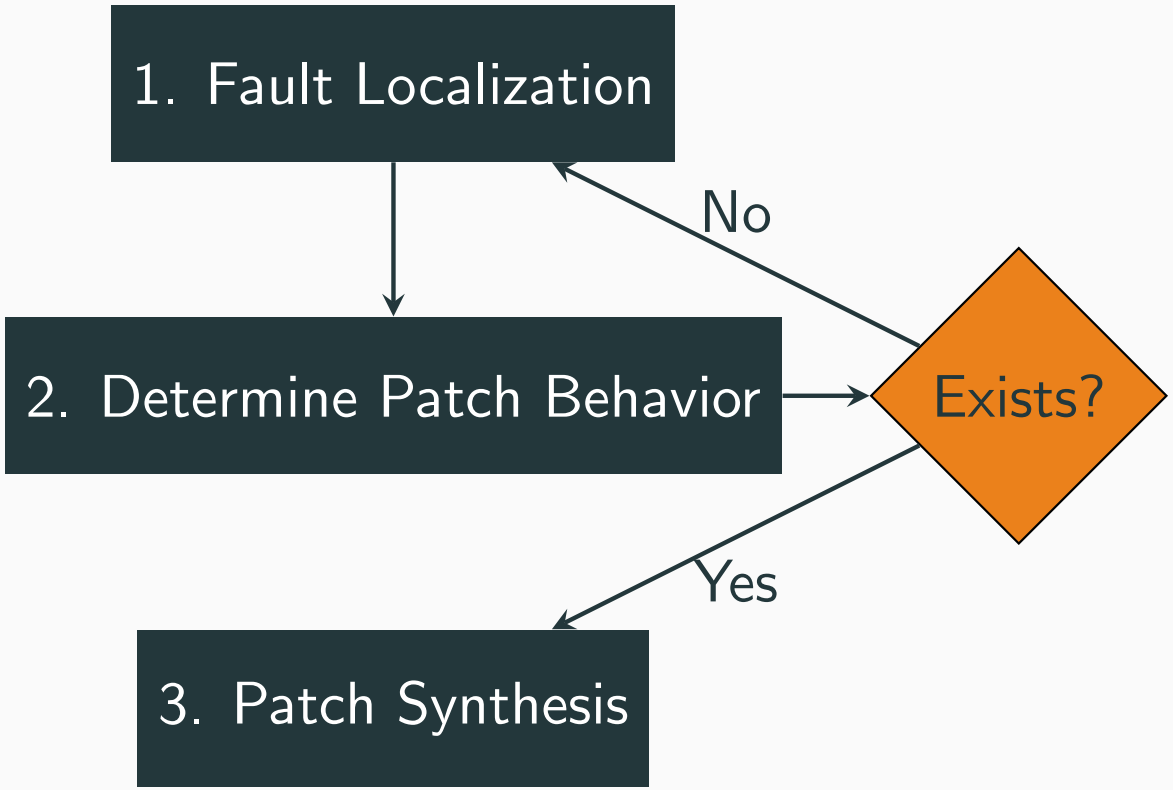
# DynaMoth

The three main steps:



# DynaMoth

The three main steps:



# Patch Behavior

**Angelic value** Chandra et al., 2011

The angelic value is the value of a condition that is required to pass the test-suite.

# Angelic Value - Example

```
pgcd(int u, int v) {
  if (u * v == 0) // overflow
    return (Math.abs(u) + Math.abs(v));
  ...
}
```

| | Tests | Program Behavior |
|---|---|---|
| ● | Test1 | `false` |
| ● | Test2 | `true` |
| ● | Test3 | `true` |
| ● | Test4 | `true` |

# Angelic Value - Example

```
pgcd(int u, int v) {
  if (true)
    return (Math.abs(u) + Math.abs(v));
  ...
}
```

| | Tests | Angelic Value True |
|---|---|---|
| ● | Test1 | `true` |
| ● | Test2 | `true` |
| ● | Test3 | `true` |
| ● | Test4 | `true` |

# Angelic Value - Example

```
pgcd(int u, int v) {
  if (false)
    return (Math.abs(u) + Math.abs(v));
  ...
}
```

| | Tests | Angelic Value False |
|---|---|---|
| ● | Test1 | `false` |
| ● | Test2 | `false` |
| ● | Test3 | `false` |
| ● | Test4 | `false` |

# Angelic Value - Example

```
pgcd(int u, int v) {
  if (angelicValue)
    return (Math.abs(u) + Math.abs(v));
  ...
}
```

| | Tests | Correct Behavior |
|---|---|---|
| ● | Test1 | `false` |
| ● | Test2 | `true` |
| ● | Test3 | `true` |
| ● | Test4 | `false` |

# Collect Runtime context

Collect the runtime context:
variables, parameters, attributes and methods.

```
public class MathUtil {
   ...
   int pgcd(int u, int v) {
      List myList = new ArrayList()
      if (u * v == 0) // buggy condition
         return Math.abs(u) + Math.abs(v)
   }
}
```

# Collect the runtime context

| Test | Runtime context of the buggy statement |
|---|---|
| Test1 | $u = 10, v = 15, myList.contains(u) = false$<br>$Math.abs(u) = 10, Math.abs(v) = 15$ |
| Test2 | $u = 0, v = 15, myList.contains(u) = false$<br>$Math.abs(u) = 0, Math.abs(v) = 15$ |
| Test3 | $u = 0, v = 0, myList.contains(u) = false$<br>$Math.abs(u) = 0, Math.abs(v) = 0$ |
| Test4 | $u = \frac{MAX\_INT}{2} + 1, v = 4, myList.contains(u) = false$<br>$Math.abs(u) = \frac{MAX\_INT}{2} + 1, Math.abs(v) = 4$ |

# Patch Synthesis

Generate new Java expressions

by combining the *runtime context* $\oplus$ *operators*

For example:

| Expression 1 | Operator | Expression 2 |
|---|---|---|
| $u$ | $==$ | $0$ |
| $u$ | $==$ | $v$ |
| $v$ | $==$ | $0$ |
| $u$ | $+$ | $v$ |
| $u + v$ | $==$ | $0$ |
| $u == 0$ | $\&\&$ | $v == 0$ |
| $u == 0$ | $\|\|$ | $v == 0$ |
| ... | ... | ... |

# Patch Synthesis

Compare the value of each Java expression to the Angelic Value

For example

| Test | `u + v == 0` | `u == 0 \|\| v == 0` | Angelic Value |
|---|---|---|---|
| Test1 | `false` | `false` | `false` |
| Test2 | `false` | `true` | `true` |
| Test3 | `true` | `true` | `true` |
| Test4 | `false` | `false` | `false` |

# Search space

The size of the search space:

- # suspicious statement
- # tests
- # variables
- # literals
- # methods
- # operators

Example: 1 suspicious statement $\oplus$ 2 tests $\oplus$ 3 variables $\oplus$ 3 literals $\oplus$ 6 methods $\oplus$ 11 operators $=$ 355 688 expressions

We need to reduce the search space!

# Search space reduction

### Expression semantics

- **Opt 1.** Inconsistent expressions ($/0$, $*0$, $||false$, $1 == 0$)
- **Opt 2.** Equivalent expressions ($a == b$ and $b == a$)

### Context based

- **Opt 3.** Ignores unused methods
- **Opt 4.** Expressions that do not evolve between tests are constant

After these search space reduction:
From 355 688 to 23 137 generated expressions (6%)

# Optimization

Optimize the synthesis by statically analysing the program:

- prioritize expressions that are more used in the program
- prioritize expressions that are present in the buggy statement

**Side effect**

the patch looks like existing code of the program

# Evaluation

# Evaluation protocol

Run DynaMoth on a dataset of real Java bugs

Compare the number of patches again the results of NOPOL (Xuan et al., 2016 TSE) which is the best tools in Java.

# Benchmark

Defects4J (Just et al., 2014) a benchmark of real Java bugs.

| Project name | # bugs |
|---|---|
| JFreeChart | 26 |
| Commons Lang | 65 |
| Commons Math | 106 |
| Joda Time | 27 |
| Closure compiler (ignored) | 133 |
| Total | 357 |

# Results

Comparison of fixed bugs

| Projects | # Fixed **DynaMoth** | # Fixed Nopol | # Total |
|---|---|---|---|
| JFreeChart | 7 | 6 | 26 |
| Commons Lang | 1 | 7 | 65 |
| Commons Math | 17 | 21 | 106 |
| Joda Time | 2 | 1 | 27 |
| Total | 27 | 35 | 224 |

With 8 bugs that are not fixed by Nopol

# Case study - Math 46

## Search space

88 expressions to combine:

- # constants: 32
- # method invocations: 52
- # field access: 2
- # variables: 2

with 11 operators

738 combined expressions (110 ms) before finding the patch

```
@@ -258,2 +258,4 @@ class Complex implements FieldElement,
    Serializable  {
  if (divisor.isZero) {
+   if (0 == this.multiply(this).getReal())
      return NaN;
+   }
```

# Related work

- **GenProg** by Forrest et al., 2009
  Comparison: DynaMoth generates patches that do not exist in the project
- **Par** by Kim et al., 2013
  Comparison: DynaMoth is not limited to predefined templates
- **SemFix** by Nguyen et al., 2013
  Comparison: DynaMoth supports a larger ingredient space during the synthesis
- **Nopol** by Xuan et al., 2016
  Comparison: DynaMoth supports a larger ingredient space

# Summary

DynaMoth is a new code synthesis engine for automatic repair using dynamic synthesis.

DynaMoth supports method calls in patches.

DynaMoth fixes 27/224 bugs with 8 unique fixes.

DynaMoth is publicly available:

`https://github.com/SpoonLabs/nopol`

Experimentation data is publicly available:

`https://github.com/tdurieux/dynamoth-experiments`

# Performances

**Table 1:** The Execution time of all Defects4J bugs.

| | Average | Median | Min | Max |
|---|---|---|---|---|
| NOPOL | 0:37:47 | 0:36:49 | 0:00:36 | 1:23:34 |
| DynaMoth | 0:38:01 | 0:36:31 | 0:01:34 | 1:28:05 |

**Table 2:** The Execution time of patched Defects4J bugs.

| | Average | Median | Min | Max |
|---|---|---|---|---|
| NOPOL | 0:05:53 | 0:01:02 | 0:00:36 | 0:44:53 |
| DynaMoth | 0:08:31 | 0:03:05 | 0:01:34 | 0:53:44 |

# References I


Chandra, S., Torlak, E., Barman, S., and Bodik, R. (2011). Angelic debugging. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 121–130. IEEE.

Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C. (2009). A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM.

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, San Jose, CA, USA. Tool demo.

# References II

Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*.

Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. (2013). Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press.

Xuan, J., Martinez, M., Demarco, F., Clément, M., Lamelas, S., Durieux, T., Le Berre, D., and Monperrus, M. (2016). Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*.