DynaMoth: Dynamic Code Synthesis for Automatic Program Repair

Thomas Durieux, Martin Monperrus

DynaMoth: Dynamic Code Synthesis for Automatic Program Repair

Thomas Durieux & Martin Monperrus
University of Lille & INRIA, France

ABSTRACT

Automatic software repair is the process of automatically fixing bugs. The Nopol repair system [4] repairs Java code using code synthesis. We have designed a new code synthesis engine for Nopol based on dynamic exploration, it is called DynaMoth. The main design goal is to be able to generate patches with method calls. We evaluate DynaMoth over 224 of the Defects4J dataset. The evaluation shows that Nopol with DynaMoth is capable of synthesizing patches and enables Nopol to repair new bugs of the dataset.

1. INTRODUCTION

Automatic software repair is the process of automatically fixing bugs. Test-suite based repair, notably introduced by GenProg [8], consists in synthesizing a patch that passes a given test suite with at least one failing test case. Test-suite based repair may use or not program synthesis to create parts of the patch. For instance, Semfix [9] uses oracle-guided component-based program synthesis [6] to synthesize patches . On the contrary, other techniques do not use synthesis, as in the case of GenProg which repairs by moving and copying existing code in the program to repair the bug, which is not synthesis per se.

The Nopol repair system [4] repairs Java code using code synthesis. It repairs bugs in conditionals: buggy if conditions and missing preconditions. Nopol uses a synthesis technique based on Satisfiability Modulo Theory (SMT), derived from oracle-guided component-based program synthesis [6]. This synthesis technique enables the system to generate patch that contains arithmetic and first order logic operators, as well as simple unary method calls (with no parameters). This is an important limitation for automatic repair. Indeed, many real-world Java patches contain complex method calls. The motivation of the work presented here is to replace the SMT-based synthesis component of Nopol by a new synthesizer that is able to generate richer patches.

We have designed a new code synthesis engine for Nopol based on dynamic exploration. The main design goal is to be able to generate patches with method calls. The engine is called DynaMoth and works as follows. At runtime, the synthesis engine stops the execution of the program under repair and explores the space of all possible expressions at a given suspicious statement. Since a buggy statement can be executed by several different test cases, the synthesis engine collects those expressions over many runs, and then infers offline the valid boolean expressions.

We evaluate DynaMoth over 224 of the Defects4J dataset. The evaluation shows that Nopol with DynaMoth is capable of repairing 27 bugs. DynaMoth is able to repair bugs that have never been repaired so far. It is capable to synthesize patches with complex operators and involving method calls with parameters.

To sum up, this paper makes the following contributions:

- DynaMoth, a dynamic synthesis algorithm for automatic repair of conditional bugs
- a publicly available implementation in Java of DynaMoth
- an evaluation of DynaMoth of 224 real bugs of open-source Java programs.

This paper is organized as follows. Section 2 discusses the context of this work. Section 3 describes the main contribution of this work: a dynamic code synthesis engine for repair. Section 4 contains the evaluation of DynaMoth based on 224 real-world bugs from Defects4J. We present our conclusions in Section 6.

2. BACKGROUND

We first briefly describe test-suite-based automatic software repair and the two classes of bugs we address.

2.1 Test-suite-based Repair

Test suite based repair uses a test suite as the specification of the correct behavior of the program. The test suite has to contain at least one failing test case which characterizes the bug to be repaired. Test-suite-based repair aims at automatically generating a patch that fixes it. Nopol [4] is an automatic repair system that is able to repair two classes of bugs: buggy IF conditions and missing preconditions.

2.2 Fault Classes

Buggy IF *Conditions.*

The first kind of bugs that Nopol targets are buggy conditions in `if-then-else` statements. Figure 1) gives an example of such a repair.

Missing Preconditions.

The second class of bugs addressed by Nopol are preconditions. A precondition is a check consisting in the evaluation of a boolean predicate guarding the execution of a statement or a block: Figure 2

AST'16, May 14-15 2016, Austin, TX, USA

ISBN 978-1-4503-4151-6/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2896921.2896931

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

Figure 1: Patch example for a buggy IF condition. The original condition with a comparison operator == is replaced by a disjunction between two comparisons. An overflow could thus be avoided.

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

Figure 2: Patch example: a missing precondition is added to avoid a null reference. The patch is specific to the object-orientation of Java.

gives an example of bug fixed by adding a precondition. Preconditions are commonly used to avoid 'null-pointer' or 'out-of-bound' exceptions when accessing array elements.

2.3 Overview of Nopol

Figure 3 describes the general algorithm of Nopol. This algorithm is composed of four main steps. The first step is the localization of suspicious statements. It uses standard spectrum based fault localization using Ochia [1]. The second step is angelic value mining, that consists in trying to identify an arbitrary value required somewhere during the execution to pass the failing test(s). As third step, input-output based code synthesis is used to generate a new Java expression which is the patch. Since Nopol repairs conditions, the synthesized expressions are boolean expressions. Finally, for patch validation, Nopol re-executes the whole test suite on the patched program.

2.4 Angelic Value Mining

The angelic value mining step determines the required value of the buggy boolean expression to make the failing test cases to pass. For buggy if-conditions, angelic value mining is done by forcing at runtime the branch of the suspicious IF statement. The condition expression is arbitrarily substituted for either `true` or `false`: the state of the program is artificially modified during its execution as it would have been by an omniscient oracle or *angel*. If all failing tests pass with the modified execution, the used value is called the angelic value and is stored for use in synthesis later.

For missing preconditions, suspicious statements are forcefully skipped at runtime. When this makes the previously failing tests pass, a precondition has to to be synthesize that returns with angelic value `false` for the failing test cases. For example, in Figure 2, let us assume that the code is specified by two tests. The first test checks the normal case when `specific != null`. The second test case, which fails, executes the buggy statement with `specific == null`. In this case, the angelic value of the precondition will be `false` for the second test case. That is the boolean expression to be synthesized as precondition should return true when `specific != null` and false otherwise.

Once an angelic value is known for all test cases, we obtain an input-output synthesis problem. The input is the context of the statement under repair (all variables in the scope), the output is the angelic value. By default, Nopol uses an SMT-based code synthesis engine implementing oracle-based component guided synthesis, this engine is called SMTSynth. This paper presents a new engine for this input-output synthesis problem.

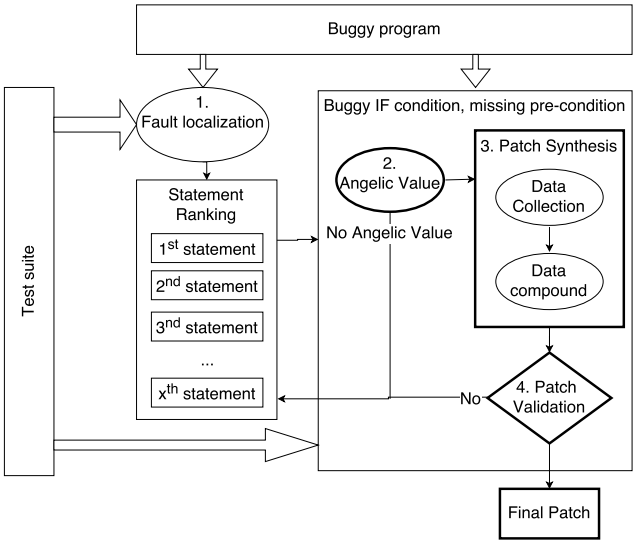

Figure 3: Overview of the Nopol repair system.

3. DYNAMIC CODE SYNTHESIS FOR AUTOMATIC REPAIR

We now present DynaMoth, a new code synthesis engine for Nopol, that is based on dynamic exploration of tentative expressions.

3.1 Code Synthesis

Thanks to angelic value mining, we know that for a set of locations and point in time during execution, a particular boolean value is expected. This boolean value is the expected result of the boolean condition to be synthesized.

As shown in Algorithm 1, the code synthesis of DynaMoth is composed of three steps: first, the collection of the runtime contexts c of the suspicious condition (see Section 3.1.1) for which it exists an angelic value. The runtime context includes parameters, variables, fields and return values of method calls; second, the generation of new boolean expressions (see Section 3.1.2); third, the comparison of each new EEXP to the expected values (see Section 3.1.3).

We now introduce two definitions, for runtime context and EEXP;

Definition (Runtime context) The runtime context of a statement is made of the values of all local variables, static variables, parameters, fields and method calls available in the scope of the statement. A runtime context also contains a reference to a test and an integer value representing the n-th times that the statement has been executed by the test.

Definition (EEXP) A Evaluated-Expression, denoted EEXP (c), is a pair e, v where e is a valid Java expression and v the value of the Java expression in a specific runtime context c.

An example of EEXP is "$e.size()$",4 which means that for a given context, the result of the evaluation of "e.size()" is 4.

3.1.1 Runtime Context Collection

Algorithm 2 presents our technique to collect the runtime context. It is achieved by stopping the execution of the program under repair at a specific location. In our case, we stop the execution only at the locations for which we have identified an angelic value

Algorithm 1 Top-level algorithm of DynaMoth

Input: A: a set of angelic values for specific statements
Output: P: a set of patches

```
1:  P ← ∅
2:  for all angelic value at statement s in A do
3:      collect simple EEXP at runtime at s (Algorithm 2)
4:      combine simple EEXP as compound EEXP (Algorithm 3)
5:      compare all EEXP against the expected angelic value (Algorithm 4)
6:      if valid EEXP found then
7:          add patch to P
8:      end if
9:  end for
10: return P
```

Algorithm 2 Collecting the set of runtime contexts for a statement S during test execution. Legend: ← means "add to set"

Input: Statement S, T: set of tests
Output: R: set of runtime contexts

```
1:  add breakpoint at S
2:  for all t in T do
3:      run t
4:      if is stopped at breakpoint then
5:          eExpList = ∅
6:          eExpList ← variables ∪ fields
7:          eExpList ← (all method calls on this)
8:          for 1 to max_depth do
9:              for all eExp in eExpList do
10:                 eExpList ← (fields eExp)
11:                 eExpList ← (all method calls on eExp)
12:             end for
13:         end for
14:         i ← iteration number for this test
15:         R_{t,i} ← eExpList
16:         proceed test execution
17:     end if
18: end for
19: return R
```

(the angelic value mining technique is described in Section 2.4.) The execution of the program is stopped using debugging technology (see Section 3.3). Once the execution is stopped, DynaMoth collects the runtime context of the statement by inspecting the variables in the scope. Then, DynaMoth collects the field values and calls the methods on each Java object in the runtime context. This step is executed recursively on the fields or the returned values that have just been collected. For example this recursion allows the synthesis of chained expressions such as `variable.method(p1, p2).getter()`. We limit the number of recursion to a constant (max_depth in Algorithm 2) in order to limit the size of the search space. For sake of performance, we do not consider possible side effects of method calls, which results in potentially corrupted collected values. This hinders completeness (some patches may not be found because of this) but not soundness because DynaMoth validates the generated patch candidate at the end of the execution as described in Section 3.1.3.

DynaMoth also collects all literals present in the method where the breakpoint is added and four literals frequently present in patches: `-1, 1, 0, null`. DynaMoth accesses all static variables and calls all static methods that are used in the scope of the suspicious statement.

Algorithm 3 Combining EEXP's with a operators.

Input: eExpList: a list of EEXP, O: a set of operators
Output: eExpList: EExp enriched with coumpound EEXP

```
1:  for 1 to max_depth do
2:      for all operator o in O do
3:          n ← number of required operators for o
4:          for all t: tuple of size n in eExpList do
5:              if types of t values compatible with o then
6:                  eExpList ← combine(t, operator)
7:              end if
8:          end for
9:      end for
10: end for
```

For each execution of a suspicious statement, DynaMoth collects a set of simple EEXP representing an expression and the result of its evaluation in the current runtime context. Those basic EEXP will be used to form the actual patch.

3.1.2 *Compound* EEXP *Synthesis*

After the collection of simple EEXP (described in Section 3.1.1), DynaMoth combines them with operators.

The first kind of compound EEXP are expression containing null checks. For example, let us consider two runtime contexts: the first contains a variable `v` which is an `array` and the second runtime context contains the same variable `v` but this time equal to `null`. With the first context, DynaMoth can access the field: `length` of the array but in the second context, `v` does not have fields. In this case, DynaMoth can generates compound EEXP of the form : `v != null && v.length == 0` evaluating to true in the first context and false in the second.

The second phase consist of generating compound expressions with binary logic and arithmetic operators. This is done by generating new expressions combining all EEXP collected so far with compatible logic/arithmetic operators: `+, -, *, /, ||, &&, ==, !=, <, <=`.[1] A combination creates a new EEXP and its value is compared to the expected value (this comparison is described in Section 3.1.3). This phase is executed recursively to create more and more complex expressions until the configuration parameter depth is reached. For example, DynaMoth can produce this type of expression: `(matrix != null && 0 == matrix.multiply(this).getReal()) && (array == null || list.getSize() < array.length)`.

3.1.3 EEXP *Validation*

Each time a boolean EEXP is created, DynaMoth verifies that the EEXP returns the expected value predicted by angelic value. If this is the case, it means that the EEXP fixes the bug locally, for this particular runtime context. When this happens, DynaMoth validates the patch for all other runtime contexts. For example, we have the EEXP`array == null || array.length == 0` and two runtime contexts: `array = null` and `array = [42],array.length = 1`. The evaluation of the EEXP with the first runtime context is equal to `true` because `array = null` and `false` with the second runtime context because `array = null` and `array.length = 1`. If those are the two expected angelic values for a precondition, it means that a patch has been found.

3.2 Optimization Techniques

The search space of the synthesis is composed of all expressions

[1] The operators `> and >=` are obtained by symmetry of `<= and <`.

Algorithm 4 Assessing whether a valid patch exists at a given statement S.

Input: eExp: a set of valued EEXP for all tests at statement s, A a set of angelic values for all tests, T a set of tests
Output: true if eExp is a valid patch

```
1:  for all t in T do
2:    for all iteration i of S in t do
3:      eExpValue ← eExp_{t,i}
4:      angelic ← A_{t,i}
5:      if eExpValue ≠ angelic then
6:        return false
7:      end if
8:    end for
9:  end for
10: return true
```

that can be generated from basic EEXP initially collected. In many cases, the search space is too large to be completely covered. Consequently, we have designed and implemented several optimizations that either reduce the size of search space or accelerate the discovery of a patch.

3.2.1 Exploration of Compound EEXP

During the synthesis of compound EEXP for a given runtime context, we only consider EEXP that have different values. For example, let us consider that DynaMoth is creating a binary expression and is looking for a boolean expression for the right operand. Furthermore, in the scope under consideration, there are two compatible variables that are both equal to `true`. In this case, DynaMoth wills only consider the first variable because the value of the new EEXP will be exactly if it had considered the second variable. This greatly prunes the search space for a single runtime context. This optimization of the search space is similar to that of J. Galenson et al. in CodeHint [5]. This does not decrease the synthesis power: if two EEXP may evaluate to the same value in one runtime context c_1 but to different value in another one c_2, the synthesis of a patch using EEXP $_1$ would be discarded in c_1 but explored in c_2. If it works on c_2 it would be validated on c_1 afterwards.

3.2.2 Number of Collected Runtime Contexts

We need a threshold on the number of collected runtime contexts. We have found that a good threshold is that DynaMoth stop collecting after 100 runtime contexts of a given suspicious statement for a given test case. For instance, if the suspicious statement is in a loop, the runtime collection will start at each loop iteration. In order to limit the execution time of the collection, DynaMoth considers only the 100 first iterations and ignores the others. This optimization is sound because the synthesized patch is still validated on the real bug with all executions of the suspicious statement.

3.2.3 Method Calls Used in Synthesized Patch

The number of possible method calls yields of huge search space. Patches are unlikely to call methods that have never been used in the program. Consequently, DynaMoth only collects method calls which are used elsewhere in the program.

3.2.4 Ignore equivalent expressions

DynaMoth ignores equivalent expressions such as binary expressions with a commutative operator (example: `x + y` and `y + x`). DynaMoth ignores binary expressions which contain a neutral operand (example: `0 + x` or multiplication by `0`). DynaMoth also ignores all trivially incorrect patches such as the comparison of two literals (example: `1 == 2`).

3.2.5 Method Invocation Time Timeout

During runtime context collection, we set a threshold on the execution time of method calls (2 seconds per default). This timeout creates an upper bound on the execution time of runtime context collection, and mitigates the problem of infinite loops caused by invalid method parameters.

3.2.6 Ordering of Runtime Contexts

All runtime contexts do not contains the same number of simple EEXP, before the combination step, as shown in Section 3.1.2. Consequently, the runtime context that contains more EEXP has more chance to produce a patch because more combinations of EEXP is possible. By considering first the runtime contexts that have more EEXP, we reduce the time spent searching the patch in a runtime context that cannot produce it.

3.2.7 Estimation of Usage Likelihood of Operators and Operands

DynaMoth collects the usage statistics of variables, operators and methods in the class that contains the suspicious statement. For instance, if we are repairing a statement in class Foo, DynaMoth extracts the information that operator "+" is used 10x in Foo, method "substring' is used 4x in Foo, etc. This usage statistics is used to drive the synthesis of compound EEXP: the composition operators that are most frequently used are prioritized during synthesis. This optimization is based on our intuition that patches are more likely to contain operators and methods that have previously been used in the class under consideration.

3.3 Implementation Details

DynaMoth uses the Java Debugging Interface as to set breakpoints and collect the runtime contexts. The Java Debugging Interface (JDI) is an API of the virtual machine that provides information for debuggers and systems needing access to the running state of a virtual machine.[2]

For collecting the usages statistics (see Section 3.2.3 and Section 3.2.7) DynaMoth uses a library for analyzing Java source code: Spoon [10]. Spoon is also used during the angelic value mining (Section 2.4) and during the final patch validation. The localization of suspicious statement is delegated to the library: GZoltar [3].

The source code of DynaMoth is part of the Nopol project [4] and is publicly available on GitHub at https://github.com/SpoonLabs/nopol

4. EVALUATION

In order to evaluate the effectiveness of DynaMoth, we execute it on the Defects4J [7] dataset of 224 real-world bugs. The methodology of this evaluation is composed of the following dimensions: the bug-fixing ability, the patch readability and the execution performance. First, our evaluation protocol is described in Section 4.1. Second, our three research questions (**RQ's**) are detailed in Section 4.2, and finally the responses to our research questions given in Section 4.3.

4.1 Protocol

We run DynaMoth to repair the bugs of the Defects4J dataset. Defects4J by Just et al. [7] is a database of 357 reproducible real

[2]See https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/. Last accessed January 2016.

Table 1: Results on the Fixability of 224 Bugs in Defects4J with Four Repair Approaches. DynaMoth repairs 27 bugs (12%).

Project	Bug Id	SMTSynth	**DynaMoth**
JFreeChart	C3	Fixed	–
	C5	Fixed	Fixed
	C6	–	Fixed
	C9	–	Fixed
	C13	Fixed	–
	C17	–	Fixed
	C21	–	Fixed
	C25	Fixed	Fixed
	C26	Fixed	Fixed
Commons Lang	**L39**	Fixed	Fixed
	L44	Fixed	–
	L46	Fixed	–
	L51	Fixed	–
	L53	Fixed	–
	L55	Fixed	–
	L58	Fixed	–
Commons Time	**T7**	–	Fixed
	T11	Fixed	Fixed
Total	43 (19.2%)	35 (15.6%)	27 (12%)

Project	Bug Id	SMTSynth	**DynaMoth**
Commons Math	**M32**	Fixed	Fixed
	M33	Fixed	Fixed
	M40	Fixed	–
	M41	–	Fixed
	M42	Fixed	Fixed
	M46	–	Fixed
	M49	Fixed	Fixed
	M50	Fixed	Fixed
	M57	Fixed	–
	M58	Fixed	Fixed
	M69	Fixed	–
	M71	Fixed	Fixed
	M73	Fixed	–
	M78	Fixed	Fixed
	M80	Fixed	–
	M81	Fixed	–
	M82	Fixed	–
	M85	Fixed	Fixed
	M87	Fixed	Fixed
	M88	Fixed	Fixed
	M96	–	Fixed
	M97	Fixed	Fixed
	M99	Fixed	Fixed
	M104	Fixed	–
	M105	Fixed	Fixed
Total	43 (19.2%)	35 (15.6%)	27 (12%)

software bugs from 5 open-source Java projects. Defects4J provides a framework which abstracts over compilation and test execution.

Defects4J is the largest open and structured database of real-world Java bugs. We use four out of five projects currently available in the Defects4J dataset, i.e., Commons Lang,[3] JFreeChart,[4] Commons Math,[5] and Joda-Time.[6] We do not use the Closure Compiler[7] project because the test suite used in this project is incompatible with our implementation.

The evaluation of DynaMoth on the 224 of Defects4J takes days to be completed and require a large amount of computer power. To overcome this problem, we use Grid5000, a grid for high performance computing [2]. We define a timeout of 1 hour and 30 minutes for each repair execution.

4.2 Research Questions

1. **RQ1**. Bug fixing ability: Which bugs of Defects4J can automatically be repaired with DynaMoth? In canonical test-suite based repair, a bug is considered as fixed when the whole test suite passes with the patched program. To answer this quantitative question, we run DynaMoth on each bug of the Defects4J dataset and count the number of fixed bugs.

2. **RQ2**. Patches Readability: Are the synthesized patches simpler than those generated by SMTSynth? The readability of the generated patches is an important factor when developers comprehend and validate them before including them in the code base. To answer this question, we look at the number of expressions (variable, literals, parameters), method calls and operators in each patch.

3. **RQ3**. Performance: How long is the execution of DynaMoth? For automatic repair to be applied in practice, it is important that it does not run too long. For instance, requiring one week to find a patch is not acceptable for developers. To answer this question, we analyze the execution time of DynaMoth when it finds a patch.

4.3 Empirical Results

We perform the experiment described in Section 4.1. The total execution time of this experiment is more than 11 days.

4.3.1 Bug fixing Ability

RQ1. Which bugs of Defects4J can automatically be repaired with DynaMoth?

Table 1 presents the bugs of the Defects4J dataset that are fixed either by SMTSynth or DynaMoth. Each line presents a bug of the dataset Defects4J and each column contains the patching result of each approach. DynaMoth is able to fix 27 (12% of 224 bugs) bugs, SMTSynth 35 (15.6%) bugs. This shows that the new synthesis engine based on dynamic synthesis is not as good as the original one of Nopol based on constraint based synthesis. In theory, DynaMoth should be able to generate all patches produced by SMTSynth, but due to the complexity of the DynaMoth implementation, there remains bugs for specific complex cases.

However, there are 8 of them are only fixed by DynaMoth. This is explained that the synthesis spaces of Nopol and DynaMoth do not completely overlap. We analytically know that some patches can be synthesized and not by SMTSynth. For instance, those containing method calls with parameters. This empirical result confirms this analysis, and is a piece of evidence of the correctness of

[3]Apache Commons Lang, http://commons.apache.org/lang.

[4]JFreeChart, http://jfree.org/jfreechart/

[5]Apache Commons Math, http://commons.apache.org/math.

[6]Joda-Time, http://joda.org/joda-time/

[7]Google Closure Compiler, http://code.google.com/closure/compiler/

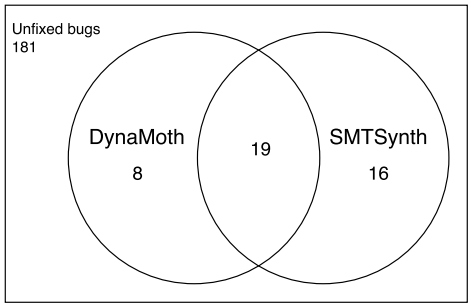

Figure 4: The figure illustrates the bugs commonly fixed by DynaMoth and SMTSynth.

our implementation. Figure 4 presents the decomposition of the 43 bugs fixed by DynaMoth and SMTSynth. And, 19 of them are fixed by both techniques.

The results of this experimentation is publicly available on GitHub.[8]

Answer to RQ1. 27 bugs of the Defects4J dataset are fixed by DynaMoth. This shows it applicability on real bugs. Among them, 8 are only fixed by DynaMoth and not by SMTSynth. This shows that SMTSynth and DynaMoth are complementary, one can try both in conjunction for repairing bugs in practice.

4.3.2 Patch readability

RQ2. Are the synthesized patches simpler than those generated by SMTSynth?

We compare the readability of patches generated by SMTSynth and DynaMoth. Table 2 shows the number of expressions (*E*) which are variables and constants, method calls (*M*) and operators (*O*) in each patch. In 9/19 cases, DynaMoth generates patches that contain less elements, hence that are easier to read. In 7/19 cases, DynaMoth generates patches that contain equals number of elements. In 3/19 cases, DynaMoth generates patches that contain more elements. In average DynaMoth contains less expressions (2.26 vs 2.94), less operators (0.85 vs 2) but more method calls (0.44 vs 0.28).

This quantitative result is confirmed by the manual analysis. The manual analysis shows that 22/27 patches synthesized by DynaMoth are easy to read and 5/27 to medium. As expected, there is a clear relation between the small amount of expressions and method calls in the patches and the readability.

Answer to RQ2. DynaMoth synthesizes patches that are simpler and easier to read compared to SMTSynth.

4.3.3 Performance

RQ3. How long is the execution of DynaMoth on one bug?

The experiment in this paper is run on a grid where the nodes are machines such as Intel Xeon X3440 Quad-core processor and 15GB of RAM.

Table 3 shows the total execution time of Nopol and DynaMoth for all bugs. Table 4 shows the execution time for the bugs which are repaired. The average execution time of all bugs are similar for DynaMoth and SMTSynth. For the repaired bugs (Table 4), DynaMoth is slower than the SMT-based synthesis of Nopol.

[8]The GitHub repository of the experimental data of DynaMoth: https://github.com/tdurieux/defects4j-repair/

Table 2: Results on the Readability of 58 Bugs in Defects4J for the four repair approaches. Legends: *E* # expressions, *M* # method calls and *O* # operators

Project	Bug Id	SMTSynth	**DynaMoth**
JFreeChart	C3	E: 2 M: 0 O: 1	–
	C5	E: 2 M: 0 O: 1	**E: 1 M: 1 O: 0**
	C6	–	E: 4 M: 0 O: 3
	C9	–	E: 2 M: 0 O: 1
	C13	E: 2 M: 0 O: 1	–
	C17	–	E: 2 M: 1 O: 0
	C21	–	E: 1 M: 2 O: 2
	C25	E: 2 M: 0 O: 1	E: 1 M: 1 O: 0
	C26	E: 2 M: 0 O: 1	**E: 1 M: 0 O: 0**
Lang	L39	E: 1 M: 0 O: 0	E: 1 M: 0 O: 0
	L44	E: 2 M: 1 O: 1	–
	L46	E: 1 M: 0 O: 0	–
	L51	E: 2 M: 0 O: 1	–
	L53	E: 5 M: 0 O: 5	–
	L55	E: 2 M: 0 O: 1	–
	L58	E: 2 M: 0 O: 1	–
Time	T7	–	E: 1 M: 0 O: 0
	T11	E: 5 M: 0 O: 5	**E: 2 M: 0 O: 1**
Math	M32	E: 8 M: 6 O: 8	**E: 0 M: 1 O: 0**
	M33	E: 2 M: 1 O: 1	**E: 1 M: 1 O: 1**
	M40	E: 6 M: 0 O: 5	–
	M41	–	E: 2 M: 0 O: 1
	M42	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M46	–	E: 2 M: 2 O: 1
	M49	E: 2 M: 0 O: 1	E: 3 M: 1 O: 1
	M50	E: 2 M: 0 O: 1	E: 2 M: 1 O: 0
	M57	E: 3 M: 0 O: 1	–
	M58	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M69	E: 3 M: 0 O: 2	–
	M71	E: 4 M: 1 O: 3	**E: 2 M: 0 O: 1**
	M73	E: 6 M: 0 O: 6	–
	M78	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M80	E: 2 M: 0 O: 1	–
	M81	E: 2 M: 0 O: 1	–
	M82	E: 2 M: 0 O: 1	–
	M85	E: 3 M: 0 O: 2	**E: 2 M: 0 O: 1**
	M87	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M88	E: 2 M: 1 O: 1	**E: 2 M: 0 O: 1**
	M96	–	E: 3 M: 0 O: 2
	M97	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M99	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M104	E: 8 M: 0 O: 6	–
	M105	E: 6 M: 0 O: 5	**E: 3 M: 1 O: 1**
Total		E: 103 M: 10 O: 70	E: 61 M: 12 O: 23
Avg.		E: 2.94 M: 0.28 O: 2	E: 2.26 M: 0.44 O: 0.85

Answer to RQ3. The execution time of DynaMoth is comparable to that of SMTSynth for all bugs but slower when we only consider the fixed bugs. However the average repair time remains acceptable for classical repair scenarios.

5. RELATED WORK

Test-suite-based repair approaches (described in Section 2.1) use a test suite as program specification to generate and validate a patch. We discuss them with respect to code synthesis.

Le Goues et al. [8] proposed GenProg, a test-suite-based repair approach using genetic programming in the C language. GenProg does no do any synthesis, it moves and copies existing code in the program. SemFix by Nguyen et al. [9] uses symbolic execution for fixing assignments and conditions, the synthesis engine is the

Table 3: The Execution time of all Defects4J bugs.

	Average	Median	Min	Max
SMTSynth	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 4: The Execution time of patched Defects4J bugs.

	Average	Median	Min	Max
SMTSynth	**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

same algorithm as SMTSynth, but DynaMoth is another completely different implementation.

The closest synthesis engine is CodeHint [5]. CodeHint [5] synthesizes Java code from runtime data inside the development environment. It uses the Java runtime Debug Interface to collect the runtime data and call methods. CodeHint uses optimizations in order to reduce the search space by ignoring expression that have a low score according to classical usage. The score of an expression is computed using with the type of operators in the expression, and the usage frequency of each method call in the expression. The goal of CodeHint (helping developers) is completely different from automatic repair. This implies a key technical difference: CodeHint has to synthesize an expression for a single runtime contexts, while DynaMoth has to generate an expression that is valid for a many different runtime contexts at the same time (the number of executions of the statement under repair summed over all test cases).

6. CONCLUSION

We have presented DynaMoth, a new patch synthesizer for Java, integrated into the Nopol toolchain. The synthesis algorithm starts by collecting the execution context of suspicious statements. The DynaMoth explores a rich search space of the combination of existing values. Several optimizations are necessary to tackle the large size of the search space.

The system has been evaluated on 224 bugs from the Defects4J dataset. DynaMoth synthesizes patches for 27 of them, incl. 8 which have never been repaired before. In total, DynaMoth fixes 27 bugs from the dataset. We consider those results are encouraging. In the future, we plan to implement additional optimizations, in particular to address the combinatorial explosion of the search space caused by methods accepting many arguments and the presence of many candidate EEXP to be used as parameters.

7. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[2] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. volume 20, pages 481–494. SAGE Publications, 2006.

[3] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: an Eclipse plug-in for testing and debugging. In *Proceedings of Automated Software Engineering*, 2012.

[4] F. Demarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT. In *CSTVA'2014*, Hyderabad, India, 2014.

[5] J. Galenson, P. Reames, R. Bodik, Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663. ACM, 2014.

[6] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[7] R. Just, D. Jalali, and M. D. Ernst. 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, July 23–25, 2014. Tool demo.

[8] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

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

[10] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Technical Report 5901, 2015.