International Journal of Research in Engineering, Science and Management
Volume-1, Issue-11, November-2018
www.ijresm.com | ISSN (Online): 2581-5792

Agent based Monitoring and Ensuring Safety in Refactoring Engines

V. Banumathy[1], A. Mary Christina[2]
[1]*Lecturer, Dept. of Computer Engineering, A. D. J. Dharmambal Polytechnic College, Nagapattinam, India*
[2]*Lecturer, Dept. of Computer Engineering, Srinivasa Subbaraya Govt. Polytechnic College, Puthur, India*

***Abstract*: Refactoring's are behavior-preserving program transformations that improve the design of a program. Refactoring engines are tools that automate the application of refactoring's: first the user chooses a refactoring to apply, then the engine checks if the transformation is safe, and if so, transforms the program. Refactoring engines are a key component of modern IDEs, and programmers rely on them to perform refactoring's. Usually, compilation errors and behavioral changes are avoided by preconditions determined for each refactoring transformation. However, to formally define these preconditions and transfer them to program checks is a rather complex task. In practice, refactoring engine developers commonly implement refactorings in an ad hoc manner since no guidelines are available for evaluating the correctness of refactoring implementations. A bug in the refactoring engine can have severe consequences as it can erroneously change large bodies of source code. We present an agent based technique to test Java refactoring engines and also informs the developer about the safety of refactoring. It automates test input generation by using JDolly, a Java program generator that exhaustively generates programs for a given scope of Java declarations. The refactoring under test is applied to each generated program. The technique uses an agent based approach for detecting behavioral changes, as an oracle to evaluate the correctness of these transformations. Finally, the technique classifies the failing transformations with the help of Bug Categorizer Agent by the kind of behavioral change or compilation error introduced by them and informs the developer about the safety of refactoring.**

***Keywords*: Refactoring, automated testing, program generation, Agent Based approach**

1. Introduction

Refactoring is the process of change a software system in such way that improves its internal structure without changing its external behavioral. Each refactoring may have preconditions that guarantee the behavioral preservation. For example, the Push down Method refactoring moves a method from the super class to the subclasses. Before we apply this change, we need to check if others methods with same signature already exist in the subclasses. Most used IDEs such as Eclipse, Net Beans, IntelliJ, and JBuilder automate a number of refactorings. They automatically check the preconditions and perform the transformation.

However, IDEs may perform incorrect transformations that introduce compilation errors or change the program behavior. Compilation errors are easier to detect; we only need to compile the refactored program. On the other hand, behavioral changes are more difficult to detect, since they are silently introduced by the tool. Currently, each IDE implements refactoring's based on an informal set of preconditions, because establishing it with respect to a formal semantics is prohibitive. An evidence of this fact is that some IDEs allow some transformations, and others do not identifying all refactoring preconditions for complex languages as Java is not trivial and formally verifying them is indeed a challenge. The current practice to avoid behavioral changes in refactoring's relies on solid tests. However, often test suites do not catch behavioral changes during transformations. They may also be refactored (for instance, rename method) by the tools since they may rely on the program structure that is modified by the refactoring. In this case, the tool changes, method invocations on the test suite and the original and refactored programs are checked against different test suites. This scenario is undesirable since the refactoring tool may change the test suite meaning. In this work, we propose a technique and algorithm (IntelRefact) for improving confidence that a refactoring is sound. It analyzes the transformation and generates unit tests suited for detecting behavioral changes. Moreover, we propose a program generator (JDolly) useful for generating inputs for testing refactoring tools. It is based on Alloy, a formal specification language, and ASTGen, an imperative framework for generating Java programs. We have evaluated Intel Refact and JDolly in two experiments. First, we evaluated Intel Refact on ten refactorings of real Java programs (from 3 to 100 KLOC) performed by developers that used refactoring tools and unit tests to guarantee the behavior preservation. Finally, we used Intel Refact and JDolly to test 12 refactoring's implemented by Eclipse 3.4.2. As result, we have detected that many transformations performed by Eclipse change program behavior. In summary, the main contributions of this paper are the following:

- A technique and tool for improving the confidence that a refactoring is sound.
- A Java program generator useful for automated testing refactoring implementations.
- An evaluation of 10 refactoring's applied to real Java programs.
- An evaluation of our approach on automated testing 12

refactoring implemented by Eclipse.

- The performance analysis between existing system and our proposed system.

2. Existing methods

Many papers have proposed many techniques and tools for this problem. A review of better technique will be discussed below:

A. Automated behavioral testing of refactoring engines [12]

They present a technique to test Java refactoring engines. It automates test input generation by using a Java program generator that exhaustively generates programs for a given scope of Java declarations. The refactoring under test is applied to each generated program. The technique uses SAFEREFACTOR, a tool for detecting behavioral changes, as an oracle to evaluate the correctness of these transformations. Finally, the technique classifies the failing transformations by the kind of behavioral change or compilation error introduced by them. They have evaluated this technique by testing 29 refactorings in Eclipse JDT, NetBeans, and the JastAdd Refactoring Tools. We analyzed 153,444 transformations, and identified 57 bugs related to compilation errors, and 63 bugs related to behavioral changes.

B. Working of safe refactor [12]

In this step, their technique evaluates the correctness of each applied transformation. For this purpose, it uses SAFEREFACTOR [12]. First, SAFEREFACTOR checks for compilation errors in the resulting program and reports those errors; if no errors are found, it analyzes the results and generates a number of tests suited for detecting behavioral changes. SAFEREFACTOR identifies the methods with matching signature (methods with exactly the same modifier, return type, qualified name, parameter types, and exceptions thrown) before and after the transformation. Next, it applies Randoop [21], a Java unit test generator, to produce a test suite for those methods. Randoop randomly generates tests for a set of methods given a time limit. The default time limit is 2 seconds. Finally, SAFEREFACTOR runs the tests before and after the transformation and evaluates the results. If results are divergent, the tool reports a behavioral change and displays the set of unsuccessful tests. Otherwise, developers have their confidence on behavior preservation improved. Assuming the programs as input, SAFEREFACTOR first identifies the methods with matching signatures on both versions: Next, it generates unit tests for those methods within a time limit of 2 seconds. Finally, it runs the test suite on both versions and evaluates the results.

C. Working of bug categorizer

The previous step may detect a number of transformations that change behavior or introduce compilation errors. Several of those failures may be caused by a single bug in the refactoring. To manually analyze all failed refactoring's in order to identify whether these errors have been caused by a single bug is both time consuming and error-prone. Next, we describe a more efficient way of classifying the failing transformations.

1) Compilation errors

They used an automatic approach proposed by Jagannath et al. [22] to classify compilation errors. It consists of splitting the failing tests based on messages from the test oracle. The goal is to group together the failing tests related to the same bug. Their approach ignores (package, class, method, or field) names within quotes. If the same refactoring is applied to two different programs, and they result in compilation error messages following the same template, a single bug is assigned to these two failures. We developed a tool to automate this grouping.

2) Behavioral changes

Additionally, they propose an approach to classify behavioral changes by analyzing each detected change based on the characteristics of each pair source program-target program. Their approach is based on a set of filters; a filter checks whether the programs follow a specific structural pattern. For example, there are filters for transformations that enable or disable overloading/overriding of a method in the target program, relatively to the source program. They defined those filters by analyzing bugs found through the use of their approach, in addition to other bug reports from refactoring engines the filters may be applied in any order. The bug category of a behavior changing transformation is then designated by the filters matched by its source and target programs. When a transformation does not fit any of these filters, conventional debugging is demanded from refactoring engine developers. The set of filters is not complete. Currently, they focus on the Java constructs supported by JDOLLY. New filters can be proposed based on additional bugs found by refactoring engine developers. Currently, the classification of behavioral changing transformations is carried out manually. The process consists of analyzing each pair of programs and testing every filter for matches.

3. Proposed frame work

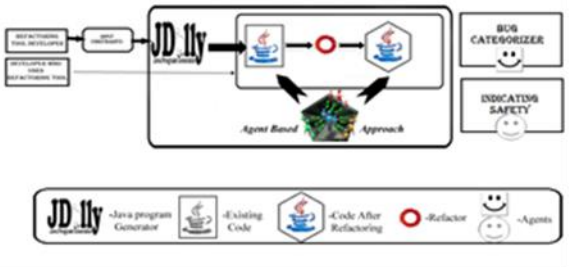

Fig. 1. Proposed architecture

In our agent based approach, the relative approach is followed. But to enhance the testing suite, we have proposed an agent based testing suite which effectively detect the behavioral changes as well as the compilation errors. And also, we have proposed an approach to classify the bugs automatically and

effectively using the BC Agent (Bug categorizer Agent). For input program generation, we are also using the JDOLLY, the JAVA program generator. The overall working of our approach is given in the following,

The overall working is given in steps as follows:

- Input program generation by JDOLLY.
- Refactoring is applied to each generated program by the refactoring tool which they want to test.
- Then we uses our agent based technique to generate the test cases and execute on the original program as well as the target program and collect the results.
- And finally, with the help of BC Agents (Our proposal), the failing transformations are classified based on the behavioral changes and compilation errors.

A. Review of JDOLLY

JDOLLY is a Java program generator that exhaustively generates programs, up to a given scope. The Alloy specification language [24] is employed as the formal infrastructure for generating programs; a Meta model for Java is encoded in Alloy, and the Alloy Analyzer finds solutions, which are translated into programs by JDOLLY, for user-specified constraints. An Alloy model or specification is a sequence of paragraphs of two kinds: signatures and constraints. Each signature denotes a set of objects associated to other objects by relations declared in the signatures. Each signature paragraph represents a type, and may declare a set of relations along with their types and other constraints on their included values.

1) Well-formed ness rules

Well-formed ness rules are specified within Alloy facts. For example a Java class cannot have two fields with the same identifier, as declared in the fact no Class Two- Fields Same Id.

```
Fact no Class Two Fields Same Id
{
all c: Class | all f1,f2: c.fields |
f1! = f2  f1.id! = f2.id
}
```

The Alloy model is then used to generate Java programs using Alloy's run commands, specifically with the generate predicate. By default, the scope of at most three objects is used for each signature. The Alloy Analyzer searches for solutions. The Alloy Analyzer does not automatically convert an Alloy instance into a Java program. In fact, we use its API to generate every possible solution. To complete the generation step, we reused the syntax tree available in Eclipse JDT for generating programs from those solutions. For example, the Alloy objects Class and Package are mapped to a type declaration and a Package Declaration, respectively. The imports are automatically calculated from each Alloy instance generated; they are included in each program. With JDOLLY, we can specify different scopes to limit program generation. For instance, if we are not interested in fields, we can specify the scope of zero. Besides, the generation can be further constrained. Suppose a context in which programs are needed with at least one class (C2) extending another one (C1) and declaring at least a method (M1); we can specify these constraints by using the following Alloy fragment. This particular specification is useful for testing the Pull Up Method refactoring, considering M1. For each instance, we pass the value given to M1 to the refactoring.

```
one sig C1, C2 extends Class {}
one sig M1 extends Method {}
pred generate[]
{C1 in C2.extend
M1 in C2.methods}
```

2) Agent based approach

We have proposed an algorithm named Intel Refact for this approach. It consists of 8 modules. They are as follows

- GUI Console.
- Monitor Agent.
- Test Case Generator Agent (TCG).
- Test Controller Agent (TC).
- Test Execution Agent(s) (TE).
- Test Collector Agent (Built-in Agent of Monitor Agent).
- Bug Categorizer Agent.
- Safety Agent.

3) Intel refact algorithm

The IntelRefact algorithm is as follows: Algorithm IntelRefact (Source_Prog, Target_Prog)

//Input: Source program and the target program (i.e., the program after refactoring)

//Output: Classification of Bugs and the information about the safety of refactoring transformations.

```
GUI(Source_Prog, Target_Prog)
{
File Src_Pro, Tar_Pro;
//Invokes Monitor Agent
MC (Src_Pro, Tar_Pro)
{
String Conditions;
File Input1, Input2, TCase;
//Invokes Test Case generator Agent
TCG (Input1, Input2)
{
recursive hybrid GE Algorithm (Input1, Input2);
return TCase;
end;
}
//Invokes Test Controller Agent
TC (TCase, Conditions,Src_Pro,Tar_Pro )
{
Generate (TCase, Conditions);
//Invokes required Test execution agents for both Source
```

Program and Target program.

```
TEx1(TCase,Src_Pro)
{
execute();
return result;
end;
}
TEx2(TCase,Src_Pro)
{
execute();
return result;
end;
}
…
…
…
TEy1(TCase, Tar_Pro)
{
 execute();
return result;
end;
}
TEy2(TCase,Tar_Pro)
{
execute();
return result;
end;
}
…
…
…
report_to_MC ();
}
//Invokes the test collector agent
TCollect (result)
{
Collect();
end TC;
Classify_Correct_Fail();
{
File Correct, Fail;
}
//Invokes the Bug Categorizer Agent.
BC(Fail)
{
Classify();
Report();
end;
}
//Invokes the Safety Agent.
Safety(Correct, Fail)
{
Indicate();
end;
}
end;
}
end;
}
end;
}
```

4) Evaluating the fitness of an individual

The fitness of an individual is evaluated as follows:

- The individual's encoding of the three parameters is decoded into three integers
- The SUT is run on these integers as parameters
- The lines of code that are executed by the SUT called on these parameters.
- The induced path is compared to the target path, and a similarity measure is computed.
- The fitness of an individual is directly proportional to how similar it is to the target path (as indicated by the similarity measure)

The hybrid algorithm presented by them outperforms an existing method in run-time. This is because it harnesses a logarithmic decay in the computational cost of the fitness function owing to the recursive classification of target paths into sub-bins. This method is better suited for testing SUTs with many paths, each of which have many constraints on them. Further, since the runs of the ESs and GAs at any given level of bin classification are independent of each other, this method is highly parallelizable. In addition, the binning allows for an approximately logarithmic decay in the number of target paths to be included in the fitness function. This implies that the fitness values of individuals in a population are computed faster as elapsed execution time progresses. This, however, is not the case for the other method, which maintains a constant-sized set of target paths throughout execution. This is one reason why the hybrid algorithm completed execution significantly faster, despite having computed significantly more fitness values. The bubble sort algorithm was used as a benchmarking SUT to illustrate this. With 64 target paths, the existing method was required to compare the path induced by each individual in every generation of the population with 64 target paths. A population size of 1000 therefore drives 64000 path comparisons per generation. However, with the hybrid, a GA is only invoked on a single target path. Thus, even with 1000 individuals in the population, only 1000 fitness evaluations are made. Further, due to the threshold values, the logarithmic nature of the decay of bin sizes forces the hybrid algorithm to perform progressively fewer fitness evaluations on every successive call to the ES on the target paths in a bin. Ultimately, the algorithm presented by them discovers inputs that induce a set of paths that contains at least all the paths induced by the inputs discovered by the other method. It is of interest to note that fewer fitness evaluations are performed by the hybrid algorithm on the Min- Max SUT. This is not anomalous. Rather, it is an artifact of the algorithm refusing to perform fitness

evaluations after an individual that induces the required target path has been discovered. Thus, since the path coverage is much higher in the case of the hybrid algorithm, there are more occasions when it stops early, explaining the lower number of fitness evaluations.

4. Experiment and results

We have evaluated our approach with refactoring transformations in Eclipse, Net Beans and on real programs. The results are tabulated in the Figure 5 and Figure 6. The performance analysis between our approach and the existing system is given in the following graph

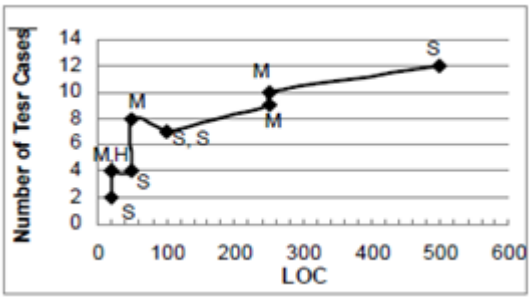

Fig. 2. Number of test cases generated

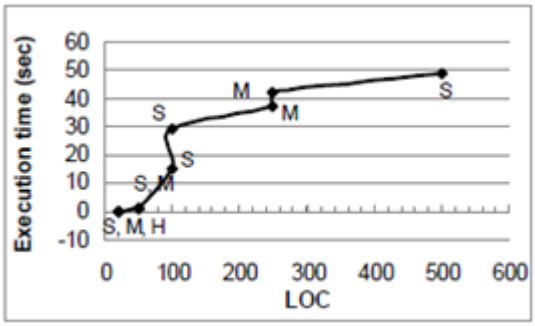

Fig. 3. Amount of time taken

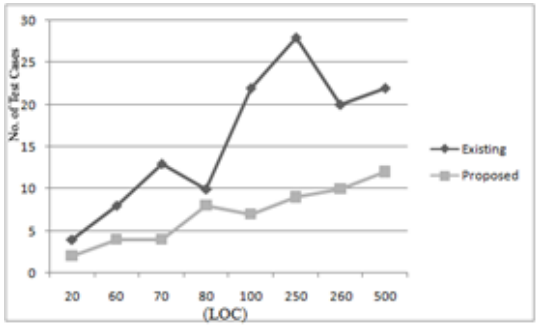

Fig. 4. Performance analysis in terms of test cases

For each refactoring, we used the same set of programs to evaluate Eclipse and Net Beans. Even though Eclipse and Net Beans have their own test suites, our technique identified unique bugs. Table 1 summarizes the bugs reported to Eclipse JDT, Net Beans, and JRRT. Our approach detects bugs related to transformation failures or weak preconditions. Our bug categorizer takes a few seconds to automatically classify all failures of a refactoring. For instance, our technique detected many compilation failures in the Push down Method refactoring implementation of Eclipse JDT. Consequently. In Eclipse JDT, the Rename Class refactoring contains three bugs; In Net Beans, three refactoring's contain four bugs each." The Rename Field, Pull Up Field, and Move Method implemented by JRRTv1 have more bugs than the similar implementation of Eclipse JDT" said by the existing system. We have founded more bugs, all related to behavioral changes. We devised an additional, automatic bug categorizer to classify these bugs. For each refactoring, it took approximately 10 minutes to automatically classify behavioral changes (depends upon LOC and bugs, so it may vary. We have presented it for maximum LOC and bugs). The number of test cases generated by us was optimized and feasible and efficient when compared with the existing system. As it is the agent based technique and it follows the approach proposed by enhancing the Efficiency of Regression testing [12] and it uses MINTS [25] tool for optimizing the test cases. The amount of time taken to generate the test cases was minimum when compared with the existing system. We have presented the algorithm and it is clear that when the agent's respectable duty, it will terminate itself. So we can maintain the trade-off between the space complexity and time complexity. So, the users need not worry about the execution of our technique or tool. We have proposed a performance analysis graph between the existing system and proposed system by From that graph, it is visualized that our existing system generates more number of test cases in minimum amount of time.

A. Bug categorizer

As it is an agent and we have feed the knowledge (i.e. filters), it can automatically classifies the bugs in minimum amount of time. It is a learning agent, so it can learn from the experience and refresh itself. It can automatically update itself. As it is an agent and we have feed the knowledge (i.e. filters), it can automatically classifies the bugs in minimum amount of time. It is a learning agent, so it can learn from the experience and refresh itself. It can automatically update itself. As it is an agent and we have feed the knowledge (i.e. filters), it can automatically classifies the bugs in minimum amount of time. It is a learning agent, so it can learn from the experience and refresh itself. It can automatically update itself. As it is an agent and we have feed the knowledge (i.e. filters), it can automatically classifies the bugs in minimum amount of time.

5. Conclusion

In this paper, we propose a technique to test Java refactoring engines. This technique is made up of JDOLLY, a Java program generator and an agent based technique and algorithm, a test suite for refactoring's and Bug Categorizer for classifying the bugs. For each refactoring, the technique generates a number of Java programs, followed by the application of the refactoring, with these programs as target. It uses our approach to evaluate

the correctness of the transformations. Finally, the technique classifies the failing transformations by kind of behavioral change or compilation error introduced by them with the help of Bug Categorizer agent (BC). We propose a Java program generator (JDOLLY [13]) to run the program generation step of our technique. It create programs for a given scope of elements (packages, classes, fields, and methods).

We have evaluated our technique by testing many refactoring's, and found many bugs related to compilation errors and behavioral changes, respectively when compared with the existing sy stem. Implementing refactoring's is not simple. Even refactoring engines written with correctness in mind, such as JRRT, still have bugs. We have demonstrated how the combination of JDOLLY and Agent based approach is powerful to detect bugs in refactoring's. In the absence of formal proofs, our technique can be useful for the improvement of previous solutions.

References

[1] Alex Groce, Gerard Holzmann, and Rajeev Joshi,"Randomized Differential Testing as a Prelude to Formal Verification", International Journal of Computer Science & Engineering Survey (IJCSES) Vol.2, No.1, Feb 2011. 2008.

[2] Amit Sharma and Miriam A. M. Capretz, "Application Maintenance Using Software Agents", IEEE, Florence, Italy, 2001, pp. 55-64.

[3] Brett Daniel Danny Dig Kely Garcia Darko Marinov,"Automated Testing of Refactoring Engines",In Proc. International Symposium on Software Testing and Analysis (ISSTA), July 2002.

[4] Carlos Pacheco Michael D. Ernst,"Randoop: Feedback-Directed Random Testing for Java"2006

[5] Chia-En Lin, Krishna M. Kavi, Frederick T. Sheldon, Kris M. Daley and Robert K. Abercrombie," A Methodology to Evaluate Agent Oriented Software Engineering Techniques", Proceedings of the 40th Hawaii International Conference on System Sciences – 2007.

[6] Cristinel Mateis, Markus Stumptner, Dominik Wieland, "JADE – AI Support for Debugging Java Programs",2006.

[7] Darko Marinov and Sarfraz Khurshid,"TestEra: A Novel Framework for Automated Testing of Java Programs"2002.

[8] David Coppit, Jinlin Yang,"Software Assurance by Bounded Exhaustive Testing",IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 31, NO. 4, APRIL 2005.

[9] Dhavachelvan, "Complexity system for software measures: towards multi agent based software testing"IEEE Transactions on SE, 2013.

[10] Gao Jing, Sch. of Comput. & Inf. Eng., Inner Mongolia Agric. Univ., Huhhot, China ; Lan Yuqing,"Agent-Based Distributed Automated Testing Executing Framework",Computational Intelligence and Software Engineering, 2009. CiSE 2009.

[11] Gordon Fraser, Andreas Zeller, "Exploiting Common Object Usage in Test Case Generation"2011.

[12] Gustavo Soares, Student Member, IEEE, Rohit Gheyi, and Tiago Massoni,"Automated Behavioral Testing of Refactoring Engines",IEEE transactions on software engineering, vol. 39, no. 2, february 2013.

[13] Hitesh Tahbildar1 and Bichitra Kalita2,"AUTOMATED SOFTWARE test data generation: direction of research", International Journal of Computer Science & Engineering Survey (IJCSES) Vol.2, No.1, Feb 2011.

[14] K. Karnavel, V. Divya, Gnanakeerthika and P. Karthika,"Agent Based Software Testing Framework (ABSTF) for Application Maintenance", IJREAT International Journal of Research in Engineering & Advanced Technology, Volume 1, Issue 1, March, 2013.

[15] T.M.S.Ummu Salima, A.Askarunisha, N.Ramaraj,"Enhancing The Efficiency Of Regression Testing Through Intelligent Agents"2008.