LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems

Ali Parsai(✉), Alessandro Murgia, and Serge Demeyer

Antwerp Systems and Software Modelling Lab, University of Antwerp, Antwerpen, Belgium
{ali.parsai,alessandro.murgia,serge.demeyer}@uantwerpen.be

Abstract. Mutation testing is a well-studied method for increasing the quality of a test suite. We designed LittleDarwin as a mutation testing framework able to cope with large and complex Java software systems, while still being easily extensible with new experimental components. LittleDarwin addresses two existing problems in the domain of mutation testing: having a tool able to work within an industrial setting, and yet, be open to extension for cutting edge techniques provided by academia. LittleDarwin already offers higher-order mutation, null type mutants, mutant sampling, manual mutation, and mutant subsumption analysis. There is no tool today available with all these features that is able to work with typical industrial software systems.

Keywords: Software testing · Mutation testing · Mutation testing tool · Complex Java systems

1 Introduction

Along with the popularity of agile methods in recent times came an emphasis on test-driven development and continuous integration [5,10]. This implies that developers are interested in testing their software components early and often [28]. Therefore, the quality of the test suite is an important factor during the evolution of the software. One of the extensively studied methods to improve the quality of a test suite is mutation testing [8].

Mutation testing was first proposed by DeMillo, Lipton, and Sayward to measure the quality of a test suite by assessing its fault detection capabilities [8]. Mutation testing has been shown to simulate faults realistically [4,17]. This is because the faults introduced by each mutant are modeled after common mistakes developers make [16]. Mutation testing is demonstrated to be a more powerful coverage criteria in comparison with data-flow, statement, and branch coverage [11,43].

Recent trends in scientific literature indicate a surge in popularity of this technique, along with an increased usage of real projects as the subjects of scientific experiments [16]. In literature, topics such as creating more robust mutants

M. Dastani and M. Sirjani (Eds.): FSEN 2017, LNCS 10522, pp. 148–163, 2017.
DOI: 10.1007/978-3-319-68972-2_10

using higher-order mutation [15,20,32,35], reducing redundancy among mutants using mutant subsumption [3,24,34], and reducing the number of mutants using mutant selection [12,13,44] are gaining popularity. Despite its benefits, the idea of mutation testing is not widely used in industry. Consequently, mutation testing research stays behind since it lacks fundamental experiments on industrial software systems. We believe that, beyond the computationally expensive nature of mutation testing [31], the reluctance of industry can stem from the shortage of mutation testing tools that can both (i) work on large and complex systems, and (ii) incorporate new and upcoming techniques as an experimental framework.

In this paper, we try to fill this gap by introducing LittleDarwin. LittleDarwin is designed as a mutation testing framework aiming to target large and complex systems. The design decisions are geared towards a simple architecture that allows the addition of new experimental components, and fast prototyping. In its current version, LittleDarwin facilitates experimentation on higher-order mutation, null type mutants, mutant sampling, manual mutation, and mutant subsumption analysis. LittleDarwin has been used for experimentation on several large and complex open source and industrial projects [36–38].

The rest of the paper is structured as follows. We provide background information about mutation testing in Sect. 2. We explain the design and the implementation of our tool in Sect. 3, and summarize the experiments that have been performed using our tool in Sect. 4. We conclude the paper in Sect. 5.

2 Mutation Testing

Mutation testing[1] is the process of injecting faults into a software system to verify whether the test suite detects the injected fault. Mutation testing starts with a *green* test suite — a test suite in which all the tests pass. First, a faulty version of the software is created by introducing faults into the system *(Mutation)*. This is done by applying a known transformation *(Mutation Operator)* on a certain part of the code. After generating the faulty version of the software *(Mutant)*, it is passed onto the test suite. If there is an error or failure during the execution of the test suite, the mutant is marked as killed *(Killed Mutant)*. If all tests pass, it means that the test suite could not catch the fault, and the mutant has survived *(Survived Mutant)* [16].

Mutation Operators. A mutation operator is a transformation which introduces a single syntactic change into its input. The first set of mutation operators were reported in King et al. [19]. These mutation operators work on essential syntactic entities of the programming language such as arithmetic, logical, and relational operators. They were introduced in the tool Mothra which was designed to mutate the programming language FORTRAN77. In 1996, Offutt et al. determined that a selection of few mutation operators is enough to produce similarly

[1] The idea of mutation testing was first mentioned by Lipton, and later developed by DeMillo, Lipton and Sayward [8]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [6].

capable test suites with a four-fold reduction of the number of mutants [29]. This reduced-set of operators remained more or less intact in all subsequent research papers. With the advent of object-oriented programming languages, new mutation operators were proposed to cope with the specifics of this programming paradigm [18,25].

Equivalent Mutants. If the output of a mutant for all possible input values is the same as the original program, it is called an *equivalent mutant*. It is not possible to create a test case that passes for the original program and fails for an equivalent mutant, because the equivalent mutant is indistinguishable from the original program. This makes the creation of equivalent mutants undesirable, and leads to false positives during mutation testing. In general, detection of equivalent mutants is undecidable due to the halting problem [30]. Manual inspection of all mutants is the only way of filtering all equivalent mutants, which is impractical in real projects due to the amount of work it requires. Therefore, the common practice within today's state-of-the-art is to take precautions to generate as few equivalent mutants as possible, and accept equivalent mutants as a threat to validity (accepting a false positive is less costly than removing a true positive by mistake [9]).

Mutation Coverage. Mutation testing allows software engineers to monitor the fault detection capability of a test suite by means of mutation coverage (see Eq. 1) [16]. A test suite is said to achieve *full mutation test adequacy* whenever it can kill all the non-equivalent mutants, thus reaching a mutation coverage of 100%. Such test suite is called a *mutation-adequate test suite*.

$$Mutation\,Coverage = \frac{Number\,of\,killed\,mutants}{Number\,of\,all\,non\text{-}equivalent\,mutants} \tag{1}$$

Higher-Order Mutants. First-order mutants are the mutants generated by applying a mutation operator on the source code only once. By applying mutation operators more than once we obtain higher-order mutants. Higher-order mutants can also be described as a combination of several first-order mutants. Jia et al. introduced the concept of higher-order mutation testing and discussed the relation between higher-order mutants and first-order mutants [14].

Mutant Subsumption. Mutant subsumption is defined as the relationship between two mutants `A` and `B` in which `A` subsumes `B` if and only if the set of inputs that kill `A` is guaranteed to kill `B` [23]. The subsumption relationship for faults has been defined by Kuhn in 1999 [21], but its use for mutation testing has been popularized by Jia et al. for creating hard to kill higher-order mutants [14]. Later on, Ammann et al. tackled the theoretical side of mutant subsumption [3]. In their paper, Ammann et al. define *dynamic* mutant subsumption, which redefines the relationship using test cases. Mutant `A` dynamically subsumes Mutant `B` if and only if (i) `A` is killed, and (ii) every test that kills `A` also kills `B`. The main purpose behind the use of mutant subsumption is to reliably detect redundant mutants, which create multiple threats to the validity of mutation testing [34]. This is often done by determining the dynamic subsumption relationship among

a set of mutants, and keeping only those that are not subsumed by any other mutant.

Mutant Sampling. To make mutation testing practical, it is important to reduce its execution time. One way to achieve this is to reduce the number of mutants. A simple approach to mutant reduction is to randomly select a set of mutants. This idea was first proposed by Acree [2] and Budd [6] in their PhD theses. To perform random mutant sampling, no extra information regarding the context of the mutants is needed. This makes the implementation of this technique in mutation testing tools easier. Because of this, and the simplicity of random mutant sampling, its performance overhead is negligible. Random mutant sampling can be performed uniformly, meaning that each mutant has the same chance of being selected. Otherwise, random mutant sampling can be enhanced by using heuristics based on the source code. The percentage of mutants that are selected determines the *sampling rate* for random mutant sampling.

3 Design and Implementation

In this section, we discuss the implementation details of LittleDarwin, and provide information on our design decisions.

3.1 Algorithm

LittleDarwin is designed with simplicity in mind, in order to increase the flexibility of the tool. To this effect, it mutates the Java source code rather than the byte code in order to defer the responsibility of compiling and executing the code to the build system. This allows LittleDarwin to remain as flexible as possible regarding the complexities stemming from the build and test structures of the target software. The procedure is divided into two phases: *Mutation Phase* (Algorithm 1), and *Test Execution Phase* (Algorithm 2).

Mutation Phase. In this phase, the tool creates the mutants for each source file. LittleDarwin first searches for all source files contained in the path given as input, and adds them to the processing queue. Then, it selects an unprocessed source file from the queue, parses it, applies all the mutation operators, and saves all the generated mutants.

```
Input  : Java source files
Output: Mutated Java source files

queue ← all Java source files;
while queue ≠ ∅ do
    srcFile ← queue.pop();
    mutants[srcFile] ← mutate(srcFile);
end
return mutants ;
```

Algorithm 1. Mutation Phase

Test Execution Phase. In this phase, the tool executes the test suite for each mutant. First the build system is executed without any change to ensure that the test suite runs "green". Then, a source file along with its mutants are read from the database, and the output of the build system is recorded for each mutant. If the build system fails (exits with non-zero status) or times out, the mutant is categorized as killed. If the build system is successful (exits with zero status), the mutant is categorized as survived. Finally, a report is generated for each source file, and an overall report is generated for the project (see Fig. 3 for an example of this).

```
Input  : Mutated Java source files
Output: Mutation Testing Report
if executeTestSuite() is successful then
    foreach srcFile do
        queue ← mutants[srcFile];
        backup(srcFile);
        while queue ≠ ∅ do
            mutantFile ← queue.pop();
            replace(srcFile,mutantFile);
            result[mutantFile] ← executeTestSuite();
        end
        restore(srcFile);
        Generate report for srcFile ;
    end
    Generate overall report;
end
return reports;
```

Algorithm 2. Test Execution Phase

3.2 Components

The data flow diagram of the main internal components of LittleDarwin is shown in Fig. 1. The following is an explanation of each main component:

JavaRead. This component provides methods to perform input/output operations on Java files. LittleDarwin uses this component to read the source files, and write the mutants back to disk.

JavaParse. This component parses Java files into an abstract syntax tree. This is necessary to produce valid and compilable mutants. To implement this functionality, an Antlr4[2] Java 8 grammar is used along with a customized version of Antlr4 runtime. Beside providing the parser, this component also provides the functionality to pretty print the modified tree back to a Java file.

[2] http://www.antlr.org/.

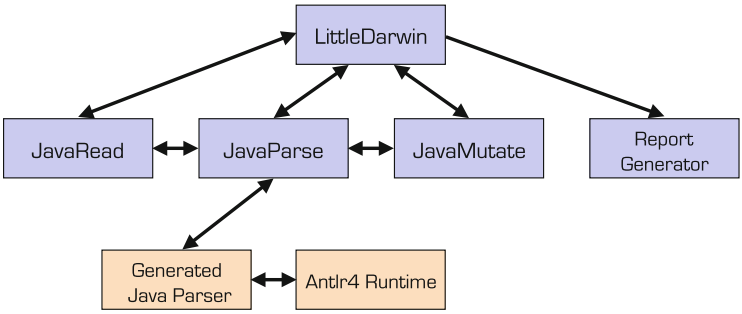
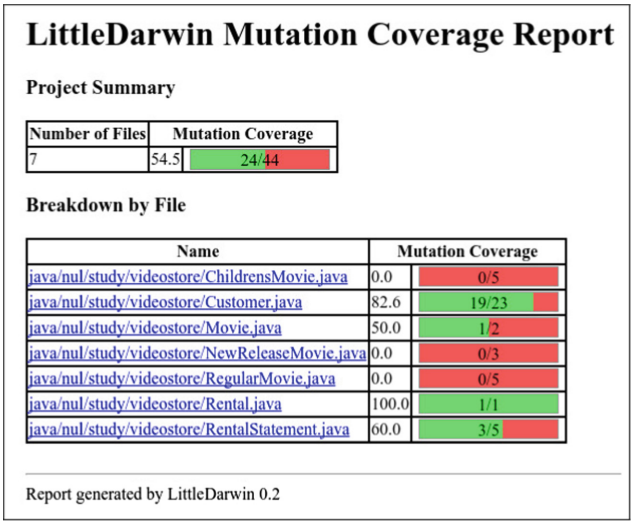


Fig. 1. Data flow diagram for LittleDarwin components

JavaMutate. This component manipulates the abstract syntax tree (AST) created by the parser. Subsection 3.3 explains the mutation operators of LittleDarwin in detail. The currently implemented mutation operators search the provided AST for mutable nodes matching the predefined patterns (for example, *AOR-B* looks for all binary arithmetic operator nodes that do not contain a string as an operand), and they perform the mutation on the tree itself. This gives the developer flexibility in creating new complicated mutation operators. Even if a mutation operator introduces a fault that needs to change several statements at once, and depends on the context of the statements, it can be implemented using a complicated search pattern on the AST. The mutation operators are designed to exclude mutations that would lead to compilation errors. However, not all of these cases can be detected using an AST (e.g. AOR-B on two variables that contain strings). Handling of such cases are therefore left for the post-processing unit that filters such mutants based on the output of the Java compiler. In order to preserve the maximum amount of information for post-processing purposes, for each mutant a commented header is created. This header contains the following information: (i) the mutation operator that created the mutant, (ii) the mutated statement before and after the mutation, (iii) the line number of the mutated statement in the original source file, and (iv) the id number of the mutated node(s). An example is shown in Fig. 2.

```
/* LittleDarwin generated mutant
 mutant type: relationalOperatorReplacement
 ----> before: daysRented   <= 0
----> after: daysRented   >0
----> line number in original file: 35
----> mutated nodes: 66
*/
```

Fig. 2. The header of a LittleDarwin mutant



LittleDarwin Mutation Coverage Report

Project Summary

Number of Files	Mutation Coverage	
7	54.5	24/44

Breakdown by File

Name	Mutation Coverage	
java/nul/study/videostore/ChildrensMovie.java	0.0	0/5
java/nul/study/videostore/Customer.java	82.6	19/23
java/nul/study/videostore/Movie.java	50.0	1/2
java/nul/study/videostore/NewReleaseMovie.java	0.0	0/3
java/nul/study/videostore/RegularMovie.java	0.0	0/5
java/nul/study/videostore/Rental.java	100.0	1/1
java/nul/study/videostore/RentalStatement.java	60.0	3/5

Report generated by LittleDarwin 0.2

Fig. 3. LittleDarwin project report (Color figure online)

Report Generator. This component generates HTML reports for each file. These reports contain all the generated mutants and the output of the build system after the execution of each mutant. In the end, an overall report is generated for the whole project (Fig. 3).

3.3 Mutation Operators of LittleDarwin

There are 9 default mutation operators implemented in LittleDarwin listed in Table 1. These operators are based on the reduced-set of mutation operators that

Table 1. LittleDarwin mutation operators

Operator	Description	Example	
		Before	After
AOR-B	Replaces a binary arithmetic operator	$a + b$	$a - b$
AOR-S	Replaces a shortcut arithmetic operator	$++a$	$--a$
AOR-U	Replaces a unary arithmetic operator	$-a$	$+a$
LOR	Replaces a logical operator	$a \,\&\, b$	$a \mid b$
SOR	Replaces a shift operator	$a >> b$	$a << b$
ROR	Replaces a relational operator	$a >= b$	$a < b$
COR	Replaces a binary conditional operator	$a \,\&\&\, b$	$a \parallel b$
COD	Removes a unary conditional operator	$!a$	a
SAOR	Replaces a shortcut assignment operator	$a *= b$	$a /= b$

were demonstrated by Offutt et al. to be capable of creating similar-strength test suites as the full set of mutation operators [29]. Since the number of mutation operators of LittleDarwin is limited, it is possible that no mutants are generated for a class that lacks mutable statements. In practice, we observed that usually only very small compilation units (e.g. interfaces, and abstract classes) are subject to this condition.

In addition to these mutation operators, there are four experimental mutation operators in LittleDarwin that are designed to simulate null type faults. These mutation operators along with the faults they simulate are provided in Table 2. We included these mutation operators based on the conclusions offered by Osman et al. [33]. In their study, they discover that the null object is a major source of software faults. The null type mutation operators are able to simulate such faults, and consequently assess the quality of the test suite with respect to them. These mutation operators cover fault-prone aspects of a method: *NullifyInputVariable* mutates the method input, *NullifyReturnValue* mutates the method output, and *NullifyObjectInitialization* and *RemoveNullCheck* mutate the statements in method body.

Table 2. Null type faults and their corresponding mutation operators

Fault	Mutation operator	Description
Null is returned by a method	NullifyReturnValue	If a method returns an object, it is replaced by `null`
Null is provided as input to a method	NullifyInputVariable	If a method receives an object reference, it is replaced by `null`
Null is used to initialize a variable	NullifyObjectInitialization	Wherever there is a `new` statement, it is replaced with `null`
A null check is missing	RemoveNullCheck	Any binary relational statement containing `null` at one side is negated

3.4 Design Characteristics

To foster mutation testing in industrial setting it is important to have a tool able to work on large and complex systems. Moreover, to allow researchers to use real-life projects as the subjects of their studies, it is also important to provide a framework that is easy to extend. In this section, we show to what extent LittleDarwin, and its main alternatives, can satisfy these requirements. As alternatives, we use PITest [7], Javalanche [41], and MuJava [27], since they are popular tools used in literature. In Table 3, we summarize the design highlights.

Compatibility with Major Build Systems. To make the initial setup of a mutation testing tool easier, it needs to work with popular build systems for Java programs. LittleDarwin executes the build system rather than integrate into it, and therefore, can readily support various build systems. In fact, the only restrictions imposed by LittleDarwin are: (i) the build system must be able

Table 3. Comparison of features in mutation testing tools

Features		LittleDarwin	PITest [1]	Javalanche [41]	MuJava [26]
Compatibility with	Maven	✓	✓	×	×
	Ant	✓	✓	×	×
	Gradle	✓	✓	×	×
	Others	✓	×	×	×
Support for complex test structures		✓	×	×	×
Optimized for performance		×	✓	✓	✓
Optimized for experimentation		✓	×	×	×
Tested on large systems		✓	✓	✓	×
Ability to retain detailed results		✓	×	×	✓
Open source		✓	✓	✓	✓

to run the test suite, and (ii) the build system must return non-zero if any tests fail, and zero if it succeeds. PITest address the challenge via integration into the popular build systems by means of plugins. At the time of writing it supports Maven[3], Ant[4], and Gradle[5]. Javalanche and MuJava do not integrate in the build system.

Support for Complex Test Structures. One of the difficulties of performing mutation testing on complex Java systems is to find and execute the test suite correctly. The great variety of testing strategies and unit test designs generally causes problems in executing the test suite correctly. LittleDarwin overcomes this problem thanks to a loose coupling with the test infrastructure, instead relying on the build system to execute the test suite. Other mutation testing tools reported in Table 3 have problems in this regard.

Optimized for Performance. LittleDarwin mutates the source code and performs the execution of the test suite using the build system. This introduces a performance overhead for the analysis. For each mutant injected, LittleDarwin demands a rebuild and test cycle on the build system. The rest of the mutation tools use byte code mutation, which leads to better performance.

Optimized for Experimentation. LittleDarwin is written in Python to allow fast prototyping [40]. To parse the Java language, LittleDarwin uses an Antlr4 parser. This allows us to rapidly adapt to the syntactical changes in newer versions of Java (such as Java 8). This parser produces a complete abstract syntax tree that makes the implementation of experimental features easier. In addition, the modular and multi-phase design of the tool allows reuse of each module independently. Therefore, it becomes easier to customize the tool according to the requirements of a new experiment. The other mutation tools work on byte code, and therefore do not offer such facilities.

[3] https://maven.apache.org/.
[4] https://ant.apache.org/.
[5] https://gradle.org/.

Tested on Large Systems. LittleDarwin has been used in the past on software systems with more than 82 KLOC [37,38]. PITest and Javalanche have been used in experiments with softwares of comparable size [39,41]. We did not find evidence that MuJava has been tested on large systems.

Ability to Retain Detailed Results. PITest and Javalanche only output a report on the killed and survived mutants. However, in many cases this is not enough. For example, subsumption analysis requires the name of all the tests that kill a certain mutant. To address this problem, LittleDarwin retains all the output provided by the build system for each mutant, and allows for post-processing of the results. This also allows the researchers to manually verify the correctness of the results. MuJava provides an analysis framework as well, allowing for further experimentation [27].

Open Source. LittleDarwin is a free and open source software system. The code of LittleDarwin and its components are provided[6] for public use under the terms of GNU General Public License version 2. PITest and MuJava are released under Apache License version 2. Javalanche is released into public domain without an accompanying license.

3.5 Experimental Features

In order to facilitate the means for research in mutation testing, LittleDarwin supports several features up to date with the state of the art. A summary of these features and their availability in the alternative tools is provided in Table 4. An explanation of each feature follows.

Table 4. Comparison of experimental features in mutation testing tools

Experimental features	LittleDarwin	PITest	Javalanche	MuJava
Higher-order mutation	✓	×	×	×
Mutant sampling	✓	×	×	✓
Subsumption analysis	✓	×	×	×
Manual mutation	✓	×	×	×

Higher-Order Mutation. This feature is designed to combine two first-order mutants into a higher-order mutant. It is possible to link the higher-order mutants to their first-order counterparts after acquiring the results.

Mutant Sampling. This feature is designed to use the results for sampling experiments. LittleDarwin by default implements two sampling strategies: uniform, and weighted. The uniform approach selects the mutants randomly with the same chance of selection for all mutants. In the weighted approach, a weight is assigned to each mutant that is proportional to the size of the class containing the mutant. The given infrastructure also allows for the development of other techniques.

[6] https://github.com/aliparsai/LittleDarwin.

Subsumption Analysis. This feature is designed to determine the subsumption relationship between mutants. For each mutant, this feature can determine whether the mutant is subsuming or not, which tests kill the mutant, which mutants are subsuming the mutant, and which mutants are subsumed by the mutant. It is also capable of exporting the mutant subsumption graph proposed by Kurtz et al. for each project [22,23].

Manual Mutation. This feature allows the researcher to use their manually created mutants with LittleDarwin. LittleDarwin is capable of automatically matching the mutants with the corresponding source files, and creating the required structure to perform the analysis. For example, this is useful in case the mutants are created with a separate tool.

4 Experiments

In this section, we provide a brief summary of the experiments we already performed using the experimental features of LittleDarwin on large and complex systems.

Mutation Testing of a Large and Complex Software System. We used LittleDarwin to analyze a large and complex safety critical system for Agfa HealthCare. Our attempts to use other mutation testing tools failed due to the complex testing structure of the target system. Due to this complexity, these tools were not able to detect the test suite. This is because (i) the project used OSGI[7] headers to dynamically load modules, and (ii) the test suite was located in a different component, and required several frameworks to work. The loose coupling of LittleDarwin with the testing structure allowed us to use the build system to execute the test suite, and thus, successfully perform mutation testing on the project. For more details on this experiment, including the specification of the target system, and the run time of the experiment, please refer to Parsai's master's thesis [36].

Experimenting Up to Date Techniques on Real-Life Projects. LittleDarwin was used to perform three separate studies using the up to date techniques reported in Table 4. We were able to perform these studies on real-life projects.

In our study on random mutant sampling, we noticed that related literature have two shortcomings [37]. They focus their analysis at project level and they are mainly based on toy projects with adequate test suites. Therefore, we evaluated random mutant sampling at class level, and on real-life projects with non-adequate test suites. We used LittleDarwin to study two sampling strategies: uniform, and weighted. We highlighted that the weighted approach increases the chance of inclusion of mutants from classes with a small set of mutants in the sampled set, and reduces the viable sampling rate from 65% to 47% on average. This analysis was performed on 12 real-life open source projects.

[7] https://www.osgi.org/developer/specifications/.

In our study on higher-order mutation testing, we used LittleDarwin to perform our experiments [38]. We proposed a model to estimate the first-order mutation coverage from higher-order mutation coverage. Based on this, we proposed a way to halve the computational cost of acquiring mutation coverage. In doing so, we achieved a strong correlation between the estimated and actual values. Since LittleDarwin retains the information necessary for post-processing the results, we were able to analyze the relationship between each higher-order mutant and its corresponding first-order mutants.

We performed a study on simulating the null type faults which is currently under peer-review. In this study, we show that mutation testing tools are not adequate to strengthen the test suite against null type faults in practice. This is mainly because the traditional mutation operators of current mutation testing tools do not model null type faults. We implemented four new mutation operators in LittleDarwin to model null type faults explicitly, and we show how these mutation operators can be operatively used to extend the test suite in order to prevent null type faults. Using LittleDarwin, we were able to analyze the test suites of 15 real-life open source projects, and describe the trade offs related to the adoption of these operators to strengthen the test suite. We also used the mutant subsumption feature of LittleDarwin to perform redundancy analysis on all 15 projects.

Pilot Experiment. We performed a pilot experiment on a real life project in order to compare LittleDarwin with two of its alternatives: PITest and Javalanche. In this experiment, we used Jaxen[8] as the subject, since it has been used before to evaluate Javalanche by its authors [42]. Jaxen has 12,438 lines of production code, and 7,539 lines of test code. Table 5 shows the results of our pilot experiment. As we can see, even though LittleDarwin creates the least number of mutants, it is still slowest per-mutant. This is mainly because PITest and Javalanche both filter the mutants prior to analysis based on statement coverage. In addition, LittleDarwin relies on the build system to run the test suite, which introduces per-mutant overhead.

Table 5. Pilot experiment results

Tool	Generated mutants	Killed mutants	Mutation coverage	Analysis time	Per-mutant time
LittleDarwin	1,390	805	57.9%	2 h 23 min 45 s	6.21 s
PITest	4,315	2,145	49.8%	1 h 13 min 13 s	1.02 s
Javalanche	9,285	4,442	47.8%	1 h 35 min 23 s	0.62 s

5 Conclusion

We presented LittleDarwin, a mutation testing framework for Java. On the one hand, it can cope with large and complex software systems. This lets

[8] http://jaxen.org/.

LittleDarwin foster the adoption of mutation testing in industry. On the other hand, the tool is written in Python and released as an open source framework, namely it enables fast prototyping, and the addition of new experimental components. From this point of view, LittleDarwin shows its keen interest in representing an easy to extend framework for researchers on mutation testing. Combining these aspects allows researchers to use real-life projects as the subjects of their studies.

In the current version, LittleDarwin is compatible with major build systems, supports complex test structures, can work with large systems, and retains lots of useful information for further analysis of the results. Moreover, it already includes the following experimental features: higher-order mutation, mutant sampling, mutant subsumption analysis, and manual mutation. Using these features, we have already performed four studies on real-life projects that would otherwise not have been feasible.

Acknowledgments. This work is sponsored by the Institute for the Promotion of Innovation through Science and Technology in Flanders through a project entitled Change-centric Quality Assurance (CHAQ) with number 120028.

References

1. Pitest. http://pitest.org/
2. Acree Jr., A.T.: On mutation. Ph.D. thesis, Georgia Institute of Technology, Atlanta (1980)
3. Ammann, P., Delamaro, M.E., Offutt, J.: Establishing theoretical minimal sets of mutants. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, pp. 21–30, March 2014
4. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), pp. 402–411. ACM, New York (2005)
5. Beck, K.: Test-Driven Development: By Example. Addison-Wesley, Boston (2003). Kent Beck Signature Book
6. Budd, T.A.: Mutation analysis of program test data. Ph.D. thesis, Yale University, New Haven (1980). aAI8025191
7. Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: Pit: a practical mutation testing tool for Java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016), pp. 449–452. ACM, New York (2016)
8. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer **11**(4), 34–41 (1978)
9. Fawcett, T.: An introduction to ROC analysis. Pattern Recognit. Lett. **27**(8), 861–874 (2006). rOC Analysis in Pattern Recognition
10. Fowler, M., Foemmel, M.: Continuous integration. Technical report, Thoughtworks (2006)
11. Frankl, P.G., Weiss, S.N., Hu, C.: All-uses vs mutation testing: an experimental comparison of effectiveness. J. Syst. Softw. **38**(3), 235–253 (1997)

12. Gligoric, M., Zhang, L., Pereira, C., Pokam, G.: Selective mutation testing for concurrent code. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013), pp. 224–234. ACM, New York (2013)
13. Gopinath, R., Alipour, A., Ahmed, I., Jensen, C., Groce, A., et al.: An empirical comparison of mutant selection approaches. Oregon State University, Technical report (2015)
14. Jia, Y., Harman, M.: Constructing subtle faults using higher order mutation testing. In: Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), pp. 249–258. Institute of Electrical & Electronics Engineers (IEEE), September 2008
15. Jia, Y., Harman, M.: Higher order mutation testing. Inf. Softw. Technol. **51**(10), 1379–1393 (2009). Source Code Analysis and Manipulation (SCAM 2008)
16. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
17. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), pp. 654–665. ACM, New York (2014)
18. Kim, S., Clark, J.A., McDermid, J.A.: Class mutation: mutation testing for object-oriented programs. In: Proceedings of Net Object Days, pp. 9–12 (2000)
19. King, K.N., Offutt, A.J.: A Fortran language system for mutation-based software testing. Softw. Prac. Exp. **21**(7), 685–718 (1991)
20. Kintis, M., Papadakis, M., Malevris, N.: Isolating first order equivalent mutants via second order mutation. In: Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 2012), pp. 701–710. Institute of Electrical & Electronics Engineers (IEEE), April 2012
21. Kuhn, D.R.: Fault classes and error detection capability of specification-based testing. ACM Trans. Softw. Eng. Methodol. **8**(4), 411–424 (1999)
22. Kurtz, B., Ammann, P., Delamaro, M.E., Offutt, J., Deng, L.: Mutant subsumption graphs. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 176–185, March 2014
23. Kurtz, B., Ammann, P., Offutt, J.: Static analysis of mutant subsumption. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–10, April 2015
24. Kurtz, B.: On the utility of dominator mutants for mutation testing. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016), pp. 1088–1090. Association for Computing Machinery (ACM), New York (2016)
25. Ma, Y.S., Kwon, Y.R., Offutt, J.: Inter-class mutation operators for java. In: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE 2002), pp. 352–363. Institute of Electrical & Electronics Engineers (IEEE) (2002)
26. Ma, Y.S., Offutt, J., Kwon, Y.R.: MuJava: an automated class mutation system. Softw. Test. Verif. Reliab. **15**(2), 97–133 (2005)
27. Ma, Y.S., Offutt, J., Kwon, Y.R.: MuJava: a mutation system for Java. In: Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), pp. 827–830. ACM, New York (2006)
28. McGregor, J.D.: Test early, test often. J. Object Technol. **6**(4), 7–14 (2007). (column)

29. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol. **5**(2), 99–118 (1996)
30. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Softw. Test. Verif. Reliab. **7**(3), 165–192 (1997)
31. Offutt, A.J., Untch, R.H.: Mutation 2000: uniting the orthogonal. In: Wong, W. (ed.) Mutation Testing for the New Century, The Springer International Series on Advances in Database Systems, vol. 24, pp. 34–44. Springer, Boston (2001). doi:10.1007/978-1-4757-5939-6_7
32. Omar, E., Ghosh, S., Whitley, D.: HOMAJ: a tool for higher order mutation testing in AspectJ and Java. In: Proceedings of the IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2014), pp. 165–170. IEEE Computer Society, Washington, DC (2014)
33. Osman, H., Lungu, M., Nierstrasz, O.: Mining frequent bug-fix code changes. In: Proceedings of the Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE 2014), pp. 343–347. Institute of Electrical and Electronics Engineers (IEEE), February 2014
34. Papadakis, M., Henard, C., Harman, M., Jia, Y., Le Traon, Y.: Threats to the validity of mutation-based test assessment. In: Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016), pp. 354–365. ACM, New York (2016)
35. Papadakis, M., Malevris, N.: An empirical evaluation of the first and second order mutation testing strategies. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW 2010), pp. 90–99. IEEE Computer Society, Washington, DC, April 2010
36. Parsai, A.: Mutation analysis: an industrial experiment. Master's thesis, University of Antwerp (2015)
37. Parsai, A., Murgia, A., Demeyer, S.: Evaluating random mutant selection at class-level in projects with non-adequate test suites. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE 2016), pp. 11:1–11:10. ACM, New York (2016)
38. Parsai, A., Murgia, A., Demeyer, S.: A model to estimate first-order mutation coverage from higher-order mutation coverage. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS 2016), pp. 365–373. Institute of Electrical and Electronics Engineers (IEEE), August 2016
39. Parsai, A., Soetens, Q.D., Murgia, A., Demeyer, S.: Considering polymorphism in change-based test suite reduction. In: Dingsøyr, T., Moe, N.B., Tonelli, R., Counsell, S., Gencel, C., Petersen, K. (eds.) XP 2014. LNBIP, vol. 199, pp. 166–181. Springer, Cham (2014). doi:10.1007/978-3-319-14358-3_14
40. Prechelt, L.: An empirical comparison of seven programming languages. Computer **33**(10), 23–29 (2000)
41. Schuler, D., Zeller, A.: Javalanche: efficient mutation testing for Java. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009), pp. 297–298. ACM, New York (2009)
42. Schuler, D., Zeller, A.: (Un-)covering equivalent mutants. In: Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST 2010), pp. 45–54. Saarland University, Saarbrucken, IEEE Computer Society, Washington, DC (2010)
43. Walsh, P.J.: A measure of test case completeness. Ph.D. thesis, State University of New York at Binghamton, Binghamton (1985)

44. Zhang, L., Gligoric, M., Marinov, D., Khurshid, S.: Operator-based and random mutant selection: better together. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013), pp. 92–102. Institute of Electrical & Electronics Engineers (IEEE), November 2013