Assertion-Driven Development: Assessing the Quality of Contracts using Meta-Mutations

Thomas Knauth
TU Dresden, Germany
thomas@se.inf.tu-dresden.de

Christof Fetzer
TU Dresden, Germany
christof.fetzer@tu-dresden.de

Pascal Felber
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

Abstract—Agile development methods have gained momentum in the last few years and, as a consequence, test-driven development has become more prevalent in practice. However, test cases are not sufficient for producing dependable software and we rather advocate approaches that emphasize the use of assertions or contracts over that of test cases. Yet, writing self-checks in code has been shown to be difficult and is itself prone to errors. A standard technique to specify runtime properties is design-by-contract (DbC). But how can one test if the contracts themselves are sensible and sufficient? We propose a measure to quantify contract (DbC). But how can one test if the contracts themselves are sensible and sufficient? We propose a measure to quantify the goodness of contracts (or assertions in a broader sense). We introduce meta-mutations at the source code level to simulate common programmer errors that the self-checks are supposed to detect. We then use random mutation testing to determine a lower and upper bound on the detectable mutations and compare these bounds with the number of mutants detected by the contracts. Contracts are considered “good” if they detect a certain percentage of the detectable mutations. We have evaluated our tools on Java classes with contracts specified using the Java Modeling Language (JML). We have additionally tested the contract quality of 19 implementations, written independently by students, based on the same specification.

Keywords: Design-by-contract, self-checks, mutation testing.

I. INTRODUCTION

Modern development methods, such as the agile methodology, have helped generalize test-driven design and, consequently, improve the general quality of software. However, while agile development methods have good budget and schedule performance, the quality of the code produced remains substandard compared to the quality of dependable software, e.g., software produced in the aerospace industry [1].

Test-driven development states that a developer first writes test cases before writing any line of code. Our hypothesis is that for producing dependable software, it would be more effective to write executable specifications, i.e., assertions first. One general goal of our work is to investigate the effectiveness of assertion-driven development and to develop tools that support this approach and in particular, further improve its effectiveness.

The earliest and most common instance of assertion-driven development is design by contract (DbC) [2]. It uses preconditions and postconditions to describe the obligations of the caller and the callee of a method. The caller has to make sure that the precondition of a method is satisfied before invoking it, and the callee has to make sure that the invariants and the postcondition of the method are satisfied before returning from it. DbC has several advantages: (1) because it provides a precise definition of the obligations by the caller and the callee, it documents the method; (2) it facilitates model checking in the sense that it helps to distinguish between correct and incorrect program behavior; (3) it amplifies test cases when the program produces a correct result but violates some assertion (i.e., an invariant, a precondition, or a postcondition) [3], [4]; and (4) it helps during the debugging of programs: the number of steps between the activation of a software fault and the violation of an assertion is typically much smaller than the latency of other symptoms like crash failures or wrong results. It also helps assign blame to either the caller (violation of the precondition) or the callee (violation of the postcondition or an invariant).

In addition to the above mentioned advantages, assertions also facilitate run-time detection of hardware and software errors. Assertions used to monitor Fortran programs date back even further than the more structured approach of DbC [5]. The online error detections of programs is often lacking [6] and our own (in-progress) experiments indicate that this results in corrupted checkpoints and incorrect results. Assertions can be used to stop the propagation of errors during run-time and to trigger recovery mechanisms like retries or rollbacks. In particular, Heisenbugs [7], i.e., seemingly random bugs that disappear on retry, can be detected with assertions and masked by repeating the computation. Bohrbugs [7], i.e., reproducible bugs, cannot be masked by simple retries; instead, they must be taken care of by other mechanism like graceful degradation. Alternatively, changing the environment of an application can also increase the likelihood that Heisenbugs, and even some Bohrbugs, do not reappear upon retry [8].

Using assertions for error detection is conceptually simple but difficult to get right in practice. Consider that our null hypothesis is that a program is correct. Assertions might produce false positives and false negatives. A false positive occurs when the program exhibits correct behavior but some assertion is violated. We say that such an assertion is unsound. A false negative occurs when the program exhibits incorrect behavior (with respect to some explicitly or implicitly given specification) but none of the assertions is violated. We say informally that our assertions are incomplete. More precisely, to define what an incomplete assertion is, one needs to consider its type and the points where it is evaluated. For a precondition to be incomplete, the caller does not satisfy its obligations (e.g., that a stack must contain at least one element before calling pop()). But the precondition is nevertheless satisfied.
For a postcondition to be incomplete, the callee does not do what it is supposed to do but the postcondition still evaluates to true on exit. Finally, an invariant defined over the state of a set of objects is incomplete if the state of at least one object in the set is inconsistent when the invariant is evaluated but no violation is reported. The notion of incomplete assertions agrees with Howden’s observation [9] that oracles are often either necessary or sufficient, but seldom both.

The incorrect contract for square_root was replaced by a new contract\(^3\) that is similar to that of Rosenblum [10] (except that < is replaced by <=). For this contract, it is more difficult to see that it is actually also unsound. The issue is that an arithmetic overflow occurs for large argument values, which invalidates the postcondition. Random testing can find a contract violation within a reasonable number of random test cases (see Figure 2).

Similarly, one can easily find contracts in the literature that are not complete. For example, Rosenblum [10] provides a contract for sorting an array (see Figure 3). The original contract was incomplete because the definition of a permutation was not completely correct. The contract permits the input values of the array to be, say, \{3, 2, 2\} and the output values \{2, 3, 3\}. The corrected contract, published in [11], solves this issue by requiring that the output values are strictly monotonically increasing (S[i] <= S[i+1] was replaced by S[i] < S[i+1]). However, there is no precondition that requires all input values to be distinct. Hence, a caller can invalidate the postcondition by passing an array that contains duplicate values like \{3, 2, 2\}. Since sorting algorithms can cope with duplicate values, the corrected postcondition is actually unsound and this can again be detected using simple random testing.

Providing a complete contract might actually not always be desirable. Contracts can be very difficult to write and comprehend. If they are also used for documentation purposes, it is typically more desirable to move implementation-oriented assertions into the code instead. In what follows, we use the term assertions to also refer to such implementation-oriented assertions and to contracts.

Contracts need not be complete for failure detection: the goal is only to detect potential faults in a given implementation and not in all possible implementations. Therefore, we define a contract to be weakly complete iff the contract detects all likely software faults of a given implementation. To define likely failures is a contentious issue: this depends on many factors like the software development process, tools, and developers. We base our work on the study of open source software [12] that shows that 65% of the discovered software bugs were simple mutation faults.

Our approach is to use random testing to detect unsound contracts and assertions. Random testing has been shown to have similar effectiveness as systematic testing approaches [13]. However, there are certain cases in which random testing is not as effective (basically, programs with statements / paths


---

**Fig. 1:** (Left.) Unsound contract for square root function. (Right.) Random testing (100 millions executions) of the contract showed that in almost all cases a contract violation is detected within only one test case.

Making sure that assertions are correct, i.e., sound and complete, is difficult. Empirical evidence indicates that it is not uncommon that in 50% of the cases the assertion/contract is faulty and not the implementation.\(^1\) In this paper, we provide empirical data on the correctness of contracts using a study involving 19 2nd year Master students.

To demonstrate the difficulty of providing correct contracts, we describe some faulty assertions from papers and systems that advertise the use of assertions and/or DbC. For example, a web page about contract programming with the D programming language\(^2\) showed the contract for a square root function depicted in Figure 1. The functions return value might need to be rounded because the return type is a long. Obviously, the postcondition is unsound, i.e., can produce false positives, because it is only satisfied by arguments for which the square root is a natural number (i.e., the return value does not need to be rounded). An interesting observation is that a violation of the postcondition can easily be detected using random testing, i.e., calling the square_root function with randomly generated arguments.


\[^1\]From the presentation of the work by Bertrand Meyer, Ilincu Ciupa, Lisa Liu, Manuel Oriol, Andreas Leitner, Raluca Borca-Muresan on “Systematic evaluation of test failure results” at the workshop on Reliability Analysis of System Failure Data, Cambridge, March 2007.

---

**Fig. 2:** (Left.) Improved contract for square root function (source: [10]). (Right.) Random testing of the contract showed that, for 10\(^6\) test cases, the probability of finding a violation is about 90%.

**Fig. 3:** Contract for sorting an array (source: [10]).
that are very unlikely to be executed). While using random testing in this work, we have also been working on tools to address this problem. However, our extended random testing tool is still in development and is not described in this paper.

We use software mutations to check the weak completeness of assertions by enumerating all likely faults in the given program. Ideally, the contracts in the program should detect all mutants: this will make sure that all likely bugs are detectable. Note that some mutants are equivalent in the sense that they will not change the program behavior. Therefore, they cannot be detected by contracts and the question we need to answer is: “Have the assertions detected all non-equivalent mutants?”

To answer this question, we define a lower and an upper bound for the weak completeness: they help to determine when the assertions in a program are “good enough.” The lower bound is defined over all likely mutants, i.e., it is the percentage of mutants detected (for a given number of test cases). The upper bound is defined over all likely mutants that are detectable, i.e., the percentage of mutants detected by the program assertions from the set of mutants that change the state of the program (in comparison to the original program). The ideal goal of the contracts of a program should be to guarantee an upper bound of 100% at the point when the lower bound has already saturated, i.e., further testing will not activate any more mutations.

We have designed and implemented a tool to perform meta-mutations for Java programs. We use it together with another tool to generate random calls to an API. Contracts and assertions are evaluated to detect unsound contracts and to compute the lower and upper bounds for the weak completeness.

We have also experimentally validated some of our hypotheses on the quality of software developed using an assertion-based methodology. To that end, we have asked a group of Master students to develop a given program individually and regularly check in their revisions in a shared repository. We have then evaluated the quality of the code and assertions of every successive revision for each student. The main lessons learned from this experiment are as follows. There exists dramatic differences in the completeness of assertions: some of the students provide assertions that are almost 99% complete while one student achieved less than 49% completeness. Our results seem to indicate that more complete contracts do provide better detection of bugs that were introduced by the students themselves. We conclude that we need tools that help to quantify and to maximize the detection capability of assertions.

The rest of the paper is organized as follows: We first discuss related work in Section II. Section III describes our approach to mutating code for checking contracts and introduces our meta-mutation tool. In Section IV, we evaluate the effectiveness of our tool on various applications and discuss our experiments with assertion-based development. Finally, Section V concludes the paper.

II._related_work

Leveson et al. showed in [3] that the effectiveness of self-checks is highly dependent on the developer who created the self-checks. In particular, in this study, among the 18 faults detected only three were found by more than one of the three participants writing self-checks for the same program. The authors also note that there is “no significant correlation between the length of a participant’s graduate or industrial experience and their success at writing self checks.” Hence, we believe it is important to have appropriate tool support to improve the quality of assertions provided by developers.

The approach of this paper is related to mutation testing [14]. The goal of mutation testing is to determine if the set of test cases detects a sufficient percentage of program mutants. The percentage of detected mutants over the number of non-equivalent mutants is called the mutation score. However, determining if a mutant is equivalent to the original program is very difficult and time consuming if it has to be done manually. Hence, some systems have tried to automate this task using constraint-based approaches (e.g., see [15]).

In this paper, we use mutations to determine if the contracts of a program are sufficiently complete. We automatically generate test cases instead of determining the adequacy of existing ones. Our belief is that contracts are more efficient to write than test cases and have a broader purpose, as they can also be used for model checking, documentation, etc. Hence, developers should use assertion-driven development instead of test-driven development: write the contracts and assertions first and only then do the coding.

We address in a pragmatic manner the problem of the equivalence of mutants. Contracts are always state-based and, hence, we only require a contract to detect mutants if, at the time the contract is evaluated, the state of the contract is different from the state the original program would be in. We conceptually transform the equivalence problem into a testing problem: we need to test the mutants to discover states that are different from that of the original program and for which the contracts, at the time of evaluation, are not violated.

We use random testing for evaluating the soundness and completeness of contracts. Random testing can be competitive with structured testing [13]. The standard testing question that we need to answer is: “When have we done sufficient testing?” In our experiments, we manually checked when our lower and upper bounds for the weak completeness are saturating. In practice, one would combine random testing with a more systematic approach like structural testing. The latter would make sure that we did not miss any important test cases, i.e., we have reached a sufficient coverage level. A recent study [16] has also shown that, despite its name, random testing seems to perform rather predictably.

For example, predicate-complete testing [17] is a good candidate for complementing random testing for mutation-based contract evaluation. Predicate-complete testing extracts all predicates from the source code and ensures that each reachable observable state (i.e., evaluation of the predicates at a
statement) of a program is covered by a test. We propose to perform random testing first and, if sufficient coverage is not achieved, one should generate test cases using constraint-based testing until a predicate-complete coverage is reached. Note that generating random test cases is more efficient because no constraints need to be solved when generating a test case. Predicate-complete testing permits computing when one has reached sufficient coverage, i.e., it has a computable test adequacy criteria.

The authors of [18] also studied the impact of design by contract using mutations. Our study provides additional data points to evaluate the impact of contracts. There are several systems that support contracts in programming languages, e.g., [19] shows how to support contracts in Java.

### III. Assessing the Quality of Contracts: The Meta-Mutation Tool

To evaluate the soundness and completeness of contracts, we have developed a meta-mutation tool that mutates Java source code. Some mutations are language independent, e.g. wrong binary operator, while others are tailored for Java, e.g. missing synchronize statements and wrong object comparisons (i.e. swapping Object.equals() with ==).

Some authors proposed to perform mutation testing on binary code [20]. We believe that source code translation is superior to instrumenting compiled code. First, it provides a finer control over the mutation process and can more accurately mimic programmer errors. For instance, some deviation has been observed for binary instrumentation between the intended mutations and those that are actually performed when working directly on compiled code (see [20]). Second, many potential mutations are likely to disappear as a result of compiler optimizations (e.g. loops may be removed, the order of statements may be altered). Finally, producing mutants in source code makes it much easier to locate and fix errors.

The tool is implemented as an Eclipse plug-in. Specifically, we use the Java Developer Tools shipped with Eclipse to do the parsing, abstract syntax tree (AST) construction and generation of mutated code. The mutator is implemented as a visitor over the AST. This design emphasized modularity and easy extensibility.

#### A. Mutation Operators

Finding the “right” mutation operators, i.e., operators that reflect typical programmer bugs, is difficult. The kind of bugs depend on course on the individual programmers but also the development tools used by the programmers. Our mutation operators loosely follow the fault classification by Durães and Madeira [12]. Their examination of various open source projects and the corresponding bug reports revealed recurring patterns for faults introduced by programmers. Our tool uses similar basic types of mutations. Table I lists some of the mutations that are currently supported, as well as the actual transformation performed in pseudo-code syntax.4

Two types of mutation operators are supported by our tool.

- **Missing constructs** model typical faults where the programmer omits one or more statements: “missing variable initialization” and “missing variable assignment” leave variables in a possibly inconsistent state; “missing else” unconditionally skips the else block, irrespective of the condition; “missing break” omits to exit the switch block after a case statement has been executed; “missing function/method call” prevents execution of a function or method.

- **Wrong constructs** model classical faults where the programmer uses wrong language features: “wrong operator” substitutes an operator for another; “wrong if/while/for/do condition” forces the condition to be inverted.

Obviously, this list of mutations is far from exhaustive, but it represents some of the most common errors encountered in real applications and provides sufficient diversity for generating the variety of mutants necessary for contract evaluation. The modular architecture of our meta-mutation tool makes it very easy to add new operators or new types of mutations (e.g., extraneous constructs), if needed.

Each mutation is guarded by a function that decides at runtime whether the mutation must be executed. A mutation is activated only if parameter id of the guard has a particular value. Typically, id uniquely identifies a mutation and activation depends on a set of active mutations, which must be initialized accordingly prior to executing the mutated code.

The Java compiler performs data flow analysis during compilation and detects errors such as uninitialized variables. Therefore, we could not blindly perform pattern matching according to the pseudo-code shown in Table I when implementing some mutations. Instead, we had to carefully ensure that the transformed code compiles without errors such as e.g. uninitialized variables.

Our mutations use three helper functions: MVI_type(), WBO_op() and WUO_op(). The MVI_type() function initializes primitive types to a default value, which can be changed during runtime, whereas object variables are initialized to null. Mutations of operations rely upon specific functions (WBO_op() and WUO_op()) to apply a given binary or unary operator to the target expression(s).

#### B. Generating and Executing Mutants

Our meta-mutation tool allows us to generate “meta-mutants,” i.e., one executable that has all possible mutations embedded. Each mutation can be independently activated at runtime. To the best of our knowledge, our tool is the first to generate such dynamic meta-mutants. MuJava [21] can also produce a single bytecode file but requires instantiation of individual mutant objects and cannot dynamically switch on and off mutations. Figures 4 and 5 shows some mutations applied to a square root function. A mutation is triggered when the constant parameter

---

4Actual transformations may slightly differ from the pseudo-code to deal with special cases (e.g., to handle break statements in the code mutated by the WIW operator).
of the guard is equal to the activation value set for the current execution. For instance, if the activation value is set to 2, then the while loop will be mutated into an if block for the current run. Note that it is also possible to activate several mutations in a given execution.

Once the meta-mutant program has been generated, we can execute it repeatedly by activating its different mutations. Each distinct run simulates a different mutant. Hence, program transformation (a costly operation) is performed only once, generating a large number of potential mutants at the same time.

IV. EXPERIMENTAL EVALUATION

Our evaluation is two-fold. We first evaluate the quality of contracts by mutating a set of mature classes developed using a DbC approach. Secondly, we study the quality of software developed by students using an assertion-based methodology.

TABLE I: Some of the mutation operators supported by our meta-mutation tool.

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Operator</th>
<th>Description</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrong object comparison</td>
<td>WOC(id, o, p)</td>
<td>The variable is initialized with a different value</td>
<td>type var = expr;</td>
</tr>
<tr>
<td>Missing break in switch</td>
<td>MBRK(id)</td>
<td>An assignment is omitted</td>
<td>if (!MBRK(id)) break;</td>
</tr>
<tr>
<td>Missing method call</td>
<td>MFC(id)</td>
<td>A switch expression</td>
<td>switch (WSE(id, expr)) switchBlock</td>
</tr>
<tr>
<td>Wrong return value</td>
<td>WRV(id)</td>
<td>Return a different value</td>
<td>return WRV(id, expr);</td>
</tr>
<tr>
<td>Wrong assignment</td>
<td>WAO(id)</td>
<td>Omit an assignment</td>
<td>return WRV(id, expr);</td>
</tr>
<tr>
<td>Wrong binary operation</td>
<td>WBO(id, expr1, expr2)</td>
<td>Return a different value</td>
<td>return WRV(id, expr1, expr2);</td>
</tr>
<tr>
<td>Wrong unary operation</td>
<td>WUO(id, expr)</td>
<td>Perform a different binary operation while preserving the expression type</td>
<td>return WRV(id, expr);</td>
</tr>
<tr>
<td>Wrong construct</td>
<td>WIC(id)</td>
<td>Wrong if condition</td>
<td>if (cond) stmt1; else stmt2;</td>
</tr>
<tr>
<td>Change if-condition</td>
<td>WIC(id)</td>
<td>Change if-condition</td>
<td>if (WIC(id, cond)) stmt1; else stmt2;</td>
</tr>
<tr>
<td>Change while-condition</td>
<td>WWC(id)</td>
<td>Change while-condition</td>
<td>while (WWC(id, cond)) stmt</td>
</tr>
<tr>
<td>Wrong switch expression</td>
<td>WSE(id, expr)</td>
<td>Change switch expression</td>
<td>switch (WSE(id, expr)) switchBlock</td>
</tr>
<tr>
<td>Wrong return value</td>
<td>WRV(id)</td>
<td>Return a different value</td>
<td>return WRV(id, expr);</td>
</tr>
<tr>
<td>Wrong construct</td>
<td>WIC(id)</td>
<td>Wrong if condition</td>
<td>if (cond) stmt1; else stmt2;</td>
</tr>
<tr>
<td>Change if-condition</td>
<td>WIC(id)</td>
<td>Change if-condition</td>
<td>if (WIC(id, cond)) stmt1; else stmt2;</td>
</tr>
<tr>
<td>Change while-condition</td>
<td>WWC(id)</td>
<td>Change while-condition</td>
<td>while (WWC(id, cond)) stmt</td>
</tr>
<tr>
<td>Wrong switch expression</td>
<td>WSE(id, expr)</td>
<td>Change switch expression</td>
<td>switch (WSE(id, expr)) switchBlock</td>
</tr>
<tr>
<td>Wrong object comparison</td>
<td>WOC(id, o, p)</td>
<td>Replace object comparison with e.g. reference comparison</td>
<td>o.equals(p)</td>
</tr>
<tr>
<td>Wrong return value</td>
<td>WRV(id)</td>
<td>Return a different value</td>
<td>return WRV(id, expr);</td>
</tr>
<tr>
<td>Wrong binary operation</td>
<td>WBO(id, expr1, expr2)</td>
<td>Perform a different binary operation while preserving the expression type</td>
<td>return WRV(id, expr1, expr2);</td>
</tr>
<tr>
<td>Wrong unary operation</td>
<td>WUO(id, expr)</td>
<td>Perform a different unary operation while preserving the expression type</td>
<td>return WRV(id, expr);</td>
</tr>
</tbody>
</table>

![Fig. 4](image-url): Original code for a square root function.

```java
double squareRoot(double x) {
    double d = x;
    double E = 1E-5;
    while (Math.abs(d - x/d) > E) {
        d = (x/d + d) / 2.0;
    }
    return d;
}
```

![Fig. 5](image-url): Mutated code of Figure 4. Only five out of twelve mutations shown in Table I are illustrated. (WBO mutation simplified and not all instances of binary operators have been mutated for clarity).

A. Setup and Protocol

**JML Classes:** In this paper, we are focusing on evaluating the completeness of contracts, i.e., their ability to detect mutants that affect the state of the application. To that end, we have picked four classes among the sample applications provided with JML [22]: a bounded stack implemented using an array (BoundedStack); a singly linked list (DLList); a doubly linked list (DLList); and an integer set implemented using a binary tree (IntegerSet). Table II shows the number of lines of code (LOC) and contract of each class, excluding comments, together with the number of mutations that our tool has inserted.

To evaluate contract completeness, we proceeded as follows. We generated random test scenarios, in which our test classes are instantiated and receive a well-formed sequence of invocations with randomly generated arguments. We have executed each scenario without activating any mutation to produce a "golden run", i.e., a reference execution assumed to be correct. It is important to consider that some mutations have no effect even when they are activated. Examples include mutated code that is never reached (dead code), mutated code that does not modify the state of the object, or mutated code that does not influence the control flow (e.g., "missing else" where if-condition is always satisfied). On the other hand, mutations that do change the program state may not be detected because the test inputs generated did not trigger the mutation. This should be kept in mind when considering the number of detectable mutations.
Therefore, we have first determined the number of detectable mutations, i.e., mutations that modify the state in at least one of the test runs (weak mutation testing). This gives an upper limit on the number of mutants that contracts can distinguish. To that end, we have compared the state of the correct object during the golden run with that of the mutant object during an identical run (i.e., same sequence of operations). If, at any step, the mutant’s behavior with respect to the internal state of the instance, the return value (if any), the exception thrown (if any), or the termination behavior (a mutation might introduce an infinite loop) was different from that of the golden run, the mutation was considered to be detectable.

We then executed the meta-mutant application by activating each of its mutations in turn. We observed which mutations were actually detected by the contracts. The percentage of mutants that lead to a contract violation gives us a lower bound on the completeness of our contracts. The upper bound on the completeness is the percentage of mutants detected by contracts among those that are detectable. Ideally, the upper bound is 100% and hence, the lower bound should be identical to the number of detectable mutations.

**Student Experiments:** We have asked a group of 19 second year Master students to solve a task from the 1997 programming contest problem set organized by the ACM International Collegiate Programming Contest [23]. Students had to simulate a multi-cast network and were asked to (a) solve the task correctly, and (b) apply basic DbC principles: pre- and post-conditions for methods, instance/class invariants, and loop invariants. Students were also asked to provide simple assertions in their code, were they see fit. Most of the students have attended a 90 minutes lecture on assertion-driven development.

Implementation was performed in Java, using JML for expressing contracts. The students stored revisions of their program, as it was being developed, in a version control system. We used this repository to study the improvement of code and contract quality.

**B. Results**

**JML Classes:** We have first evaluated the number of mutations added by our tool to various classes to observe the correlation between code size and number of mutants. The objective is to derive a function for predicting the number of mutations based on the lines of code.

We have applied our meta-mutation tool to a set of 558 JML classes and computed the number of mutations introduced per file. Comments (which include contracts) were removed because the ratio of JML annotations to Java code lines was as high as 3:1 (see Figure 6) for files with up to 200 lines of code.

As one can observe in Figure 7, there is a clear linear correlation between lines of code and number of mutations starting from around 200 LOC. Deviations stem from files which contain a rather unusual large number of simple assignment statements and function calls, as is typical for unit testing classes. The linear relationship contradicts earlier results which found a quadratic relationship between LOC and number of mutants. This is best explained by the limited set of mutation operators used in this study.

A closer examination on the small files (see Figure 8) shows that a fair number of them have no mutation at all: they are likely interfaces and, hence, only contain declarations with no executable code that could be mutated. Other small files include classes with abstract methods and constants, which explains the variance in the number of mutation.

Regarding the practicality of our tool: it took 197 seconds to process those 558 classes. Compiling them with Sun’s Java compiler took seven seconds which is an increase of two seconds compared to the compile time before introducing the meta-mutations. Measurements were taken on a Pentium M system with 2GHz, 1GB RAM running Ubuntu. The increase in byte code size due to mutation is summarized in Table III on a per package basis.

For the rest of our evaluation we focus on the four aforementioned classes from the JML package.

Figure 9 shows the percentage of detectable mutations for our four test classes. This is actually a lower bound for the number of detectable mutations because a larger number of test cases might improve that value even further. One can see that the number of detectable mutations can vary quite substantially. This implies that it would not make sense to just require a certain constant threshold when checking the completeness of contracts.

Figure 10 shows the number of mutations that were reached during random testing and that could thus be activated. As can be seen, the coverage varies for the different classes. Especially, if the mutation coverage is low (as in case of class IntegerSet), it might be useful to combine the random testing with an automated structural testing method to guarantee sufficient testing coverage and, hence, mutation coverage.

Note that, in our experiments, the difference between the mutation coverage and the percentage of detectable mutations is approximately between 25% and 40% (after 100,000 test cases).

Our upper bound for the completeness is defined as the percentage of detected mutations over the maximum percentage of detectable mutations that we have observed. Figure 11 depicts the upper bound for three of four classes. SLList approaches 80% while for BoundedStack and IntegerSet it remains below 60%. This means that the completeness of the contracts for these two classes would need to be improved.

Figure 12 gives a summary of the results for three representative classes, i.e., the lower bound and upper bound for the completeness, the mutation coverage and the percentage of

<table>
<thead>
<tr>
<th>Class</th>
<th>LOC</th>
<th>Lines of contract</th>
<th>Mutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundedStack</td>
<td>126</td>
<td>66</td>
<td>22</td>
</tr>
<tr>
<td>SLList</td>
<td>142</td>
<td>294</td>
<td>72</td>
</tr>
<tr>
<td>DLList</td>
<td>141</td>
<td>169</td>
<td>47</td>
</tr>
<tr>
<td>IntegerSet</td>
<td>200</td>
<td>101</td>
<td>127</td>
</tr>
</tbody>
</table>

**TABLE II:** Lines of code (LOC), lines of contract code, and number of mutations of our test classes.
detectable mutations.

Table IV compares the number of mutations that are detectable and indeed detected by our tool in comparison to mutations detected by the unit tests provided with the classes. It shows that for three classes the unit tests detect about the same number of mutations as our approach. The mutation coverage for class IntegerSet by our random tests were substantially lower than that of the unit tests: less than 80% in comparison to the 100% mutation coverage by the unit tests. Therefore, a combination of random testing with a more systematic testing method to improve the coverage seems to be appropriate (at least for this class). For the same mutation coverage, assertions seem to be able to detect at least the same percentage of mutations as the given unit tests.

**Student Experiments:** To successfully complete this task, the students’ program had to pass a set of test cases unknown to the students. The test cases were run whenever the students checked in their programs in the version control system. While the students were asked to check in their changes as often as possible, the number of revisions varied quite dramatically (see Figure 13).

The size of the implementation, contracts, and comments varied widely too (see Figure 14). The code size (without assertions, comments and white spaces) was between 6971 and 15073 characters while the contract size varied from 852 and 5593 characters.

We tested again the student programs using random testing (see Figure 15). While all student programs passed an automated acceptance test, several of them still produced incorrect results and false positives when they were later evaluated further. Furthermore, some of the contracts still had a very high false positive rate. Note that a high false positive rate can result in a high completeness of assertions.

We have measured the evolution of the correctness, false positive rate and false negative rate: we plot the values depending on the revision of a program (see Figures 16, 17 and 18 respectively). To be able to average the values of programs with different numbers of revisions, we repeated the results of
demonstrated: Total number of program revisions for individual students.

![Size of Contracts/Codes/Comments (last revision)](image)

**Fig. 14:** Size of the code, contracts and comments for each student.

![False positives and correct outputs](image)

**Fig. 15:** Random testing (50 test cases) of student programs shows incorrect results and false positives for assertions.

![Correctness averaged over 19 program versions](image)

**Fig. 16:** The correctness of programs typically increases with its maturity.

![False positives and correct outputs - averaged over 19 programs](image)

**Fig. 17:** Evolution of the false positive rate with increasing revision numbers.

![False negatives and contracts - averaged over 19 programs](image)

**Fig. 18:** Evolution of the false negative rate with increasing revision numbers.

### Package

<table>
<thead>
<tr>
<th>Package</th>
<th>Original (in byte)</th>
<th>Mutated (in byte)</th>
<th>Mutations</th>
<th>Inc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>org/jmlspecs/samples/list/iterator</td>
<td>531</td>
<td>531</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>org/jmlspecs/samples/list</td>
<td>1453</td>
<td>1600</td>
<td>5</td>
<td>1.10</td>
</tr>
<tr>
<td>org/jmlspecs/samples/diuserinfo</td>
<td>1498</td>
<td>1932</td>
<td>6</td>
<td>1.29</td>
</tr>
<tr>
<td>org/jmlspecs/samples</td>
<td>1825</td>
<td>2048</td>
<td>12</td>
<td>1.12</td>
</tr>
<tr>
<td>org/jmlspecs/samples/lang</td>
<td>15124</td>
<td>18784</td>
<td>87</td>
<td>1.24</td>
</tr>
<tr>
<td>org/jmlspecs/samples/reader</td>
<td>17774</td>
<td>22365</td>
<td>112</td>
<td>1.29</td>
</tr>
<tr>
<td>org/jmlspecs/samples/jmltutorial</td>
<td>25823</td>
<td>32070</td>
<td>151</td>
<td>1.24</td>
</tr>
<tr>
<td>org/jmlspecs/samples/jmlrefman</td>
<td>24380</td>
<td>31942</td>
<td>175</td>
<td>1.31</td>
</tr>
<tr>
<td>org/jmlspecs/samples/table</td>
<td>45919</td>
<td>56562</td>
<td>250</td>
<td>1.23</td>
</tr>
<tr>
<td>org/jmlspecs/samples/misc</td>
<td>51282</td>
<td>64972</td>
<td>325</td>
<td>1.27</td>
</tr>
<tr>
<td>org/jmlspecs/samples/stacks</td>
<td>47544</td>
<td>60058</td>
<td>325</td>
<td>1.26</td>
</tr>
<tr>
<td>org/jmlspecs/samples/dbc</td>
<td>35136</td>
<td>46969</td>
<td>335</td>
<td>1.34</td>
</tr>
<tr>
<td>org/jmlspecs/samples/jmlkluwer</td>
<td>51085</td>
<td>64677</td>
<td>335</td>
<td>1.27</td>
</tr>
<tr>
<td>org/jmlspecs/samples/stacks/nodes</td>
<td>53470</td>
<td>67371</td>
<td>356</td>
<td>1.26</td>
</tr>
<tr>
<td>org/jmlspecs/samples/list/nodes</td>
<td>58963</td>
<td>73821</td>
<td>388</td>
<td>1.25</td>
</tr>
<tr>
<td>org/jmlspecs/samples/sets</td>
<td>39099</td>
<td>51573</td>
<td>396</td>
<td>1.32</td>
</tr>
<tr>
<td>org/jmlspecs/samples/list/node/2</td>
<td>64571</td>
<td>81245</td>
<td>419</td>
<td>1.26</td>
</tr>
<tr>
<td>org/jmlspecs/models/tests</td>
<td>19789</td>
<td>31751</td>
<td>584</td>
<td>1.60</td>
</tr>
<tr>
<td>org/jmlspecs/samples/peml/design</td>
<td>139481</td>
<td>174419</td>
<td>825</td>
<td>1.25</td>
</tr>
<tr>
<td>org/jmlspecs/samples/list/list1</td>
<td>135640</td>
<td>170518</td>
<td>912</td>
<td>1.26</td>
</tr>
<tr>
<td>org/jmlspecs/samples/list/list2</td>
<td>151663</td>
<td>190466</td>
<td>1017</td>
<td>1.26</td>
</tr>
<tr>
<td>org/jmlspecs/samples/list/list3</td>
<td>151190</td>
<td>190033</td>
<td>1017</td>
<td>1.26</td>
</tr>
<tr>
<td>org/jmlspecs/models/resolve</td>
<td>172512</td>
<td>217107</td>
<td>1022</td>
<td>1.26</td>
</tr>
<tr>
<td>org/jmlspecs/samples/digraph</td>
<td>172198</td>
<td>220110</td>
<td>1113</td>
<td>1.28</td>
</tr>
<tr>
<td>org/jmlspecs/jmlkluwerstrategies</td>
<td>406333</td>
<td>500733</td>
<td>2409</td>
<td>1.23</td>
</tr>
<tr>
<td>org/jmlspecs/models</td>
<td>1005436</td>
<td>1349137</td>
<td>9958</td>
<td>1.34</td>
</tr>
</tbody>
</table>

### TABLE II: Increase in byte code size due to the mutations on a per package basis (sorted by number of mutations).

<table>
<thead>
<tr>
<th>Class</th>
<th>Detected</th>
<th>Detectable</th>
<th>Coverage</th>
<th>Unit tests</th>
<th>Unit test coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundedStack</td>
<td>19/42</td>
<td>30/42</td>
<td>49/42</td>
<td>18/42</td>
<td>40/42</td>
</tr>
<tr>
<td>SLList</td>
<td>50/72</td>
<td>70/72</td>
<td>47/72</td>
<td>34/72</td>
<td>60/72</td>
</tr>
<tr>
<td>DLList</td>
<td>28/47</td>
<td>38/47</td>
<td>46/47</td>
<td>28/47</td>
<td>47/47</td>
</tr>
<tr>
<td>IntegerSet</td>
<td>50/127</td>
<td>54/127</td>
<td>10/127</td>
<td>86/127</td>
<td>12/127</td>
</tr>
</tbody>
</table>

### TABLE IV: The Detected, Detectable and Coverage columns show the values after 10^3 random test cases. The Unit tests column shows the number of detected mutations using the unit tests that came with JML for those classes. The last column shows the mutation coverage of the unit tests.

The last revision of a program.

We can see that percentage of test cases passed averaged over all student programs quickly increases with the revisions (see Figure 16). Since some of the programs were not entirely correct, we never actually reach 100% of passed test cases.

The false positive rate initially drops with an increasing number of revisions (see Figures 17). However, since not all student program eventually contained correct contracts (as mentioned above, the students programs only had to pass an automated testing system to finish this task), the false positive rate actually does not approach 0%. The false negative rate also decreases with the number of revisions (see Figures 18). However, since some of the last revisions still produce incorrect outputs and the contracts are not complete (see below), the false negatives do not converge to 0% either.

To evaluate the students’ contract quality, we mutated the programs and evaluated the contracts’ completeness, i.e., how far they are able to detect non-equivalent mutants. Figure 19 shows the number of mutants we had to evaluate for individual student programs. As before, there is large variance in the number of mutants but not as strong a correlation between code size (see Figure 20) and number of mutants as one would expect.

The most important measurement is the ability of the contracts to detect software bugs. Figure 21 shows that contracts have quite a different power in detecting effective mutations. Some contracts detect non-equivalent mutants very well, which indicates that contracts can indeed be used to detect software mutations in an effective way. In particular, from our data there does not appear to be (good) correlation between the size of
contracts (see Figure 14) and the completeness of contracts. In other words, it seems that one can write compact contracts to detect mutations. Also, since there is a large variance in assertion quality, we believe that it is very important to have tools that evaluate assertion quality (recall that these contracts were developed without support of such a tool).

While detecting mutations is useful, it is far more important to detect real software bugs in a program, i.e., to have contracts with low false negative rates. Since we cannot move the assertion of a program from one revision to another (e.g., assertions might refer to variables that do not yet exist in earlier versions), there is no easy way to evaluate the final assertions of a program with earlier revisions. To give an indication how well contracts work to detect software bugs, we instead graphed the false negative rate of three selected students over all revisions that successfully compiled (see Figures 22, 23 and 24). We can see that students 4 and 9, who have low completeness (see Figure 21), seem to also have a high false negative rate over the evolution of their program. Student 8, who has a very low incompleteness, has a (comparatively) good false negative rate.

C. Discussion

In our experiments, we have seen that there is a significant difference in contract (in)completeness. Also, we have seen that simple unit tests that we used for the acceptance of student submission was insufficient: our subsequent random testing showed both incorrect outputs and assertions with false positives. Hence, we believe a tool that tests the soundness and the completeness of assertions could prove to be very helpful. Our measurements also indicate that there is a correlation between the completeness of assertions and the capabilities to detect real software bugs. A more detailed investigations of this correlation would be interesting. However, this is technically very difficult: the final assertions of a program cannot just be used for earlier revisions because, for example, variables used in contracts might not yet exist or their semantics might have changed.

Our results regarding the JML classes show that we need to achieve good mutation coverage. Random testing was appropriate for three of four JML classes that we investigated but we could not increase coverage for the fourth one. Only a high mutation coverage will result in a good measure for detectable mutations and for the upper bound we have introduced.

V. Conclusion

Design by contract (DbC) and assert-driven development in general are powerful tools for developing dependable software. Unlike standard test cases used at development time, contracts can be used as self-checks during run-time to detect software bugs, including non-reproducible Heisenbugs. Although conceptually simple, writing contracts that are both sound and complete can be very difficult. In this paper, we evaluate the use of assertions to detect mutants. To that end, we have designed a tool that performs meta-mutations to Java and C code trying to mimic typical programmer errors. The mutations we use are based on a previous study of software bugs. Our experiments and evaluations show that contracts created without tool support show quite a variance in the completeness and soundness. Our tool uses a lower and an upper bound on the number of mutants detected, which can help to determine when contracts are good enough to be trusted. Our evaluation on a set of programs using JML as a DbC specification language and our student experiments strengthen our belief that such a tool is indeed needed.

REFERENCES

Fig. 19: Number of mutations of the last revision of a student.

Fig. 20: Evolution of the number of mutations with increasing revision numbers for selected student programs.

Fig. 21: Contracts have different power to detect mutations.

Fig. 22: Student 4: False positives, false negatives and correct outputs for all revisions.

Fig. 23: Student 8: False positives, false negatives and correct outputs for all revisions.

Fig. 24: Student 9: False positives, false negatives and correct outputs for all revisions.


