Exploiting Common Object Usage in Test Case Generation

Gordon Fraser
Saarland University – Computer Science
Saarbrücken, Germany
fraser@cs.uni-saarland.de

Andreas Zeller
Saarland University – Computer Science
Saarbrücken, Germany
zeller@cs.uni-saarland.de

***Abstract*—Generated test cases are good at systematically exploring paths and conditions in software. However, generated test cases often do not make sense. We adapt test case generation to follow *patterns of common object usage,* as mined from code examples. Our experiments show that generated tests thus (a) reuse familiar usage patterns, making them easier to understand and (b) focus on common usage, thus respecting implicit preconditions and avoiding meaningless tests.**

***Keywords*-test case generation; specification mining; readability**

I. Introduction

When generating test cases for object oriented software, sequences of method calls are either assumed to be provided by the user, or determined automatically using symbolic techniques or evolutionary search. Such machine-generated sequences are well suited to test robustness or contract violations, and it is even possible to automatically add regression oracles. All these generated tests are poised towards *automatic fulfillment of a testing goal* such as coverage—but they are not meant to be read by humans. In fact, *understanding* a generated test case can be very difficult simply because these machine-generated sequences do not make sense.

As an example, consider Figure 1—a typical generated test for one of the branches of the `DateTime` class in the Joda-Time library. Yes, it covers the branch in the `minus` method. Yes, it sets up a number of objects and calls to reach that very branch. But what is the expected outcome of this test? What would be a suitable assertion that could serve as oracle for this test case? While automated exploration of method sequences will plug together any values and methods that contribute to achieve the goal, the user is left with a high number of generated executions that are hard to understand, but for which the expected outcome still must be specified.

One might argue that generated test cases are not meant for functional testing, but for *robustness testing*—that is, have the run-time system detect faults such as dereferenced null pointers, boundary or type violations, or arithmetic exceptions. But even with this restriction, generated test cases cause trouble—because methods can have *implicit preconditions* of which the developer might be aware but the test case generation tool is not. For example, many methods in the Joda-Time library take parameters of abstract base

```
int var0 = −77;
DateTime var1 = new DateTime(var0, var0, var0, var0, var0,
    var0, var0);
Instant var2 = GJChronology.DEFAULT_CUTOVER;
Date var3 = var2.toDate();
DateTime var4 = var2.toDateTime(var3);
DateTime var5 = var4.minus(var0);
```

Figure 1. Example test case generated for a branch in method `minus` in `DateTime` using search only driven by coverage.

```
long var0 = 197;
DateTimeZone var1 = DateTimeZone.UTC;
DateTime var2 = new DateTime(var0, var1);
int var3 = −91;
DateTime var4 = var2.minus(var3);
```

Figure 2. A test for the same branch exploiting object usage information.

classes or even of type `Object`—what is a useful input to such methods? As another example, passing `null` as parameter will often reveal potential faults which are not interesting as such a scenario will not happen in practice—such a test case is essentially a kind of false positive. And again, we have to assess hard-to-understand test cases manually—dozens to thousands of them.

In this paper, we present an approach to alleviate these problems. Rather than generating test cases entirely by random, we leverage *common object usage*—patterns of object interaction as found in manually written code—to make generated test cases *more similar to existing client code.* By following these implicit conventions, we address the two central problems of generated tests: We make them more readable, and we avoid violations of implicit preconditions. To demonstrate the benefit of our approach, consider Figure 2. It shows a test case that was derived for the same branch as the test case in Figure 1—but is based on common object usage found in the Joda-Time library and its test cases. Intuitively, the test case in Figure 1 relies on less functionality and should thus be easier to understand.

Summarizing, the contributions of this paper are:

API Usage Models: We analyze the source code of the software under test, its existing test cases, and any avail-

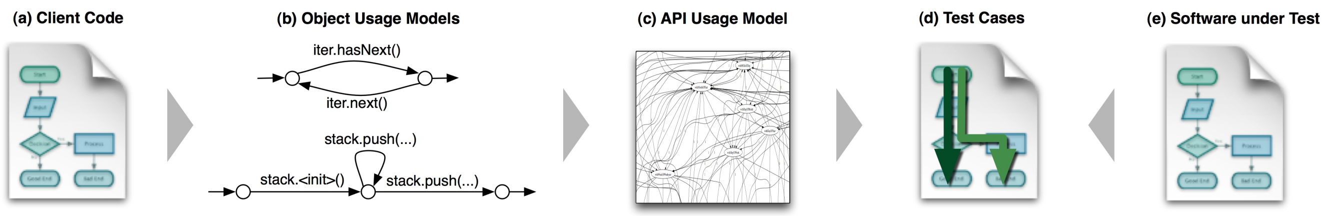

Figure 3. Our approach in a nutshell. From client code (a), we extract object usage models (b) that reflect the usage of the software under test. These models are merged into API models (c), representing the usage of the entire API. We then derive test cases (d) that conform to the API model and thus cover typical usage in the software under test (e).

able client code of the API (application programmer interface), and derive a model based on Markov chains usage models (MCUM) that represents the common usage of each of the API's classes as well as their interactions.

Test generation based on common usage: Based on the usage model, we define test generation operators for search based testing that generate test cases that resemble real code.

Our work is in line with previous research on testing with usage information (Section II), but represents an entirely new process in structural test generation. Figure 3 illustrates the overall approach: First, we extract usage models from code examples such as existing test cases or client programs, and generate an *API Usage Model* from this information (Section III). This model is used to drive the test case generation in producing coverage test suites (Section IV). As we show in Section V, this usage information has an observable effect on the resulting test suites. Even if the model information is not complete, testers can seamlessly regulate between readability and exploration, and allow uninformed search where no usage information is available.

II. Background

Usage models have been incorporated into black-box testing techniques before: Whittaker and Thomason [1] proposed the use of Markov chain usage models for statistical testing. Since then, usage models have for example been used for model based testing [2], [3], GUI testing [4], and web testing [5].

Usage models are often generated manually, possibly supported by dedicated languages [6] and tools [7], but can also be generated using systematic methods [8], [9], or mined from different artifacts such as log files [10].

In the context of structural testing based on source code, Sayre and Poore [11] describe test generation for C++ templates, where manually written usage models are annotated with test code. Thummalapenta et al. [12], [13] mined method sequences from source code with the goal to increase coverage, which is different to our goal of improving readability of test cases. Other than that, we are not aware of any work that would include usage information when deriving test cases from source code.

Structural test generation is commonly based either on symbolic techniques such as (dynamic) symbolic execution (e.g., [14], [15]) on search-techniques [16]. The prime application of such automatically generated test cases is to find possible program crashes [17] or contract violations [18]. In the context of test generation for object-oriented code, different techniques to derive method sequences have been proposed, including several search-based techniques [19]–[22].

In this paper, we propose a search-based approach to derive test cases for object-oriented software based on usage models. In a different context, meta-heuristic search techniques have already been applied to extract tests from Markov models [23].

III. Common Object Usage Analysis

In order to analyze class usage, we define a class as a set of methods $C = \{m_1, m_2, \ldots, m_n\}$, where for the sake of simplicity we treat constructors and field accesses as methods. A method $m_i = (R, P)$ is defined by its return type $R \in \mathcal{C}$, where $\mathcal{C}$ is the domain of classes, and $P = \langle p_1, p_2, \ldots, p_n \rangle$ is the (possibly empty) set of parameters, $p_i \in \mathcal{C}$. Without loss of generality, we treat primitive datatypes as elements in $\mathcal{C}$, as is common in many programming languages.

A. Mining Temporal Properties of Object Usage

The first step in analyzing common object usage is to *mine* existing source code for usage examples. We use the tool JADET [24] to mine object usage information from Java bytecode. JADET uses *temporal properties* to express relationships between individual functions. To extract such temporal properties, JADET uses two main steps:

- **Mining object usage models:** An object usage model is a finite state automaton that shows how an object "flows" through various events in a method.
- **Extracting temporal properties:** Temporal properties provide a succinct and easy-to-manipulate representation of how objects are used within the object usage models.

```
public DateTime plusMinutes(int minutes) {
  if (minutes == 0) {
    return this;
  }
  long instant =
        getChronology().minutes().add(getMillis(), minutes);
  return withMillis(instant);
}
```

Figure 4. Method in `DateTime` class.

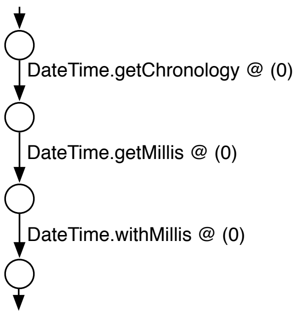

Figure 5. Object usage models for the `DateTime` object in Figure 4.

JADET creates an object usage model for each statically identifiable object used within a method. These objects are: formal parameters of methods (including the implicit `this` parameter), objects created via `new`, return values of method calls, values read from fields, and explicit constants (such as `null` and `"OK"`).

Figure 4 shows a code snippet taken from the `DateTime` class in Joda-Time. This snippet contains an instance of `DateTime` (`this`), and the return values of the method calls `getChronology`, `minutes`, and `withMillis`. Figure 5 shows the object usage model derived for `this` in the example, where the notation @(0) means the object was used as parameter 0 (callee) of the method call.

A temporal property is an ordered pair of events a and b associated with the same object, where $a \prec b$ represents an ordering where event a may happen before event b. An event associated with an object is one of the following:

- A method call (including constructor calls) with the object being used as callee or argument: `var3 = var2.toDateTime(var3)` is an event associated with `var2` and `var3`.
- A method call with the object being the value that was returned: `var3 = var2.toDate()` is such an event associated with `var3`.
- A field access; e.g., `var2 = GJChronology.DEFAULT_CUTOVER` is an event associated with `var2`.

Figure 6 shows the temporal properties derived from the object usage model shown in Figure 5.

DateTime.getChronology @ (0) $\prec$ DateTime.getMillis @ (0)
DateTime.getMillis @ (0) $\prec$ DateTime.withMillis @ (0)

Figure 6. Temporal properties for the DateTime object usage model in Figure 5.

Temporal properties are the main unit of usage information we consider when deriving usage models. We therefore define a temporal property $T = (A_1, A_2)$ as a pair of method calls, where $A_i = (C, m, p)$ is a triple with $C \in \mathcal{C}$, $m \in C$, and $0 \leq p \leq |P|$ for $m = (R, P)$.

B. Class Usage Models

Each object usage model describes how one particular instance of an object was used, and temporal properties make the relationships between pairs of events on such objects explicit. As our aim is to *produce* object instances adhering to common usage, we extract the usage information in terms of Markov chain usage models (MCUM) [1]. A Markov chain is a sequence of random variables $\{X_t\}$ with the Markov property $P[X_{t+1} = y | X_t = x_t, ..., X_0 = x_0] = P[X_{t+1} = y | X_t = x_t]$, and can be thought of as a directed, weighted graph. We define how each class is used in isolation as its *class usage model*:

Definition 1 (Class Usage Model): A class usage model $U_C = (V, E, P)$ for class $C = \{m_1, m_2, \ldots, m_n\}$ is a Markov chain consisting of the set of vertices $V \subseteq C$, the set of edges $E = V \times V$, and transition probabilities $P = E \rightarrow [0, 1]$.
Each $v \in V$ represents one of the methods of the class, and the probability of an edge between v_1 and v_2 represents the probability that method v_2 is executed after v_1. The class usage model does not need to be fully connected.

Generating class usage models from a set of temporal properties is done by iterating over all properties. If one of the method calls in a temporal property is not in the model, then a new vertex is added to V, and a new edge with weight 1 is added between the two vertices. If the edge already exists, the weight is increased by 1. After iterating over all temporal properties, the weights of the edges are normalized such that the sum of weights of all outgoing edges for a state equals to 1.

C. API Usage Models

The class usage model represents common usage sequences of class C, but for test case generation we also need information on how classes interact with each other. To this extent, we need to combine the class usage models into a single *API Usage Model*, which contains the information how the individual classes of an API interact with each other:

Definition 2 (API Usage Model): An API usage model for the set of classes $\mathcal{C}$ is a directed, weighted graph $M_{\mathcal{C}} = (V, E_C, E_P, P_C, P_P)$, where $V \subseteq \bigcup_{C \in \mathcal{C}} C$ is the set of all methods with usage information. $E_C = V \times V$

Algorithm 1 Generating an API Usage Model $M_\mathcal{C}$ from a set of temporal properties $\mathcal{T}$.

```
Require: Set of temporal properties $\mathcal{T}$
$M_\mathcal{C} \leftarrow (V, E_C, E_P, P_C, P_P)$
for $((c_1, m_1, p_1), (c_2, m_2, p_2)) \in \mathcal{T}$ do
    $V \leftarrow V \cup \{m_1, m_2\}$
    if $c_1 = c_2$ then
        $E_C \leftarrow E_C \cup (m_1, m_2)$
        $P_C((m_1, m_2)) \leftarrow P_C((m_1, m_2)) + 1$
    end if
    $E_P \leftarrow E_P \cup (m_2, p_2, m_1)$
    $P_P((m_2, p_2, m_1)) \leftarrow P_P((m_2, p_2, m_1)) + 1$
end for
normalize weights in $E_C$ and $E_P$
```

is the set of class usage edges with $\forall (v_1, v_2) \in E_C : v_1 \in C \rightarrow v_2 \in C$, and $E_P = V \times V$ is the set of parameter usage edges. $P_C = E_C \rightarrow [0, 1]$ maps class usage edges to their probabilities, and $P_P = E_P \times \mathbb{N} \rightarrow [0, 1]$ maps parameter usage edges with their parameter number to their probabilities.

Intuitively, the API Usage Model is a superset of all the class models for a set of classes $\mathcal{C}$ with an additional set of edges E_P and attached information (P_P).

Again, an API Usage Model can be generated from a set of temporal properties by iterating over this set, as shown in Algorithm 1: For each temporal property, vertices are added for both method calls. If a property describes class usage, then an edge with weight 1 is added to E_C if there is no such edge yet, else its weight is increased by 1. In addition, an edge is added to E_P for the parameter of the target method (p_2), or its weight is updated.

IV. Test Case Generation using API Usage Models

In general, test case generation from usage models is a stochastic process, where the next transition is chosen according to the probabilities of the transitions outgoing from the current state. In the case of an API usage model, this process needs to be refined to accommodate for the forward (class usage) and backward (parameter usage) edges of the model.

A. *Usage driven Test Cases*

We assume that testing should focus on one particular class C, and therefore select one of the methods of C as initial state. A test case $t = \langle m_1, m_2, \ldots, m_n \rangle$ is a sequence of method calls. The forward exploration adds new method calls of C to t according to the API Usage Model (or rather, the subpart that constitutes the class usage model of C). This process is illustrated in Algorithm 2.

When adding a new method call to the test case, its parameters need to be satisfied. If t contains an object o

Algorithm 2 GenerateTest(C)

```
Require: Set of classes $\mathcal{C}$
Require: API Usage Model $M_\mathcal{C} = (V, E_C, E_P, P_C, P_P)$
$t \leftarrow \langle\rangle$
M$\leftarrow$ choose method in $C$ randomly
while not done do
    AddCall ($t$,M)
    M'$\leftarrow$ select stochastically such that (M , M') $\in E_C$
    M $\leftarrow$ M'
end while
return  $t$
```

Algorithm 3 AddCall(T, M)

```
Require: API Usage Model $M_\mathcal{C} = (V, E_C, E_P, P_C, P_P)$
1:  for each parameter $p$ of M do
2:      if $t$ has object $o$ such that last call N: (N,$p$,M) $\in E_P$ then
3:          use $o$ for this parameter
4:      else
5:          select method N: (N,$p$,M) $\in E_P$ stochastically
6:          $o$ $\leftarrow$AddCall ($t$,N)
7:          use $o$ for this parameter
8:      end if
9:  end for
10: $t \leftarrow t$.M
11: return  return value of M
```

created as a return value or by a constructor, such that the last method call on this object o is a successor edge of the current state in E_P, then this object can be used as a parameter. If no suitable object exists, then an object can be generated by inserting a method call that is an outgoing edge for the parameter in E_P to t. This backward exploration has to be performed until the target object is created, at which point the originally chosen method call of C can be appended to the test case. This process is illustrated in Algorithm 3.

By construction, any sequence generated with this algorithm consists only of pairs of method calls that were also observed in the code examples. However, the combination of such behaviors can lead to exploration of new behavior not part of the mined examples.

Depending on the completeness of the usage information, AddCall may fail on trying to generate an object, or it might create disproportionally long sequences due to the recursion. To counter this effect, a limit on the recursion depth can be used – however, this cannot overcome the problem of missing information in M_C.

B. *Integrating Usage Information into Search-based Testing*

While the test generation approach described above can generate any number of test cases of any length resembling common object usage, the aim of achieving readable and understandable test cases requires to focus on a small set of representative test cases. To this extent, code coverage (e.g., branch coverage) is often used to control which and how many test cases are generated in practice.

A coverage criterion defines a set of goals that a test suite should satisfy. Given such a coverage goal, test cases can be generated using a number of different approaches. Meta-heuristic search techniques have been suggested as a possible solution to automate test case generation [16]. Search-based techniques have been applied to test object oriented software using method sequences [19], [22], [25] or strongly typed genetic programming [20], [21]. Following the method sequence approach [19], [22], [25], we define search operators that allow evolutionary testing of classes based on common object usage.

A genetic algorithm is an evolutionary testing technique where a set of candidate solutions is evolved using genetics-inspired operations towards satisfying a given objective. In principle, it works as follows:

1) Generate initial population (usually randomly)
2) Generate new generation as follows:
    - Determine fitness of current population
    - Select parents
    - Create offspring by crossover of parents and mutation
3) Repeat (2) until stopping criterion holds

Different objective functions have been defined in the context of test case generation; in this paper, we aim at branch coverage and use the traditional *approach level* and *branch distance* approach [16], which estimates the distance of a test case to executing a target branch.

In the method sequence approach, a chromosome is a list of method calls $\langle m_1, m_2, \ldots, m_n \rangle$ just like defined in the previous subsection. Based on this representation, we only need to define operators for mutation and for crossover to allow search-based test generation [19] to make use of object usage information.

When aiming to achieve code coverage, an incomplete usage model can affect the exploration ability. Therefore, we relax the requirement that test cases have to consist of only observed behavior to requiring that test cases reflect observed behavior *as well as possible*. In practice, this simply means we allow non-observed behavior with a low probability, both when selecting a method call to generate a parameter object and to select the next method call.

C. Mutation

Mutation is an essential part of evolutionary search, where individuals are changed independently of the population. Tonella [19] defined a set of mutation operators for method sequences, which need to be adapted to make sure that the resulting test cases still conform to the API usage model. We distinguish three main types of mutation operators:

Insertion: This operator adds a new statement to an existing test case $t = \langle m_1, \ldots, m_n \rangle$, and is illustrated in Algorithm 4. First, a position i is randomly chosen within the range $[0, n]$. Then, an object defined by one of the methods $m_1, \ldots, m_i$ is selected randomly, and the last method call on this object up to position i is determined. Then, a new method call is selected stochastically from the usage model based on this last method call.

Algorithm 4 MUTATIONINSERT(t)

```
Require: API Usage Model $M_\mathcal{C} = (V, E_C, E_P, P_C, P_P)$
1: $i \leftarrow$ random position $[0,\text{length}(t)]$
2: $m_o \leftarrow$ random method in $t$ within $[0, i]$
3: $m_l \leftarrow$ last method call in $t$ on object represented by $m_o$ in range $[0, i]$
4: select method $n : (m_o, n) \in E_C$ stochastically
5: ADDCALL($t, n$)
```

Change: This operator changes an existing method call in one of several ways; for example, it may change parameter objects with other valid, existing objects, changes primitive data types, or replaces a method call with another method call. A replacement method call has to satisfy three conditions: (1) there has to be a usage transition from the previous method call on the same object to the replacement method in the API usage model, (2) there has to be a usage transition from the replacement method to the next method call on the same object, and (3) for each parameter transition that exists in the test case for the old method, there has to be a suitable replacement parameter transition to the new method, or to an alternate method in the test case. Again, with a certain probability these conditions can be relaxed to allow exploration of uncommon behavior.

Deletion: When selecting a method call for deletion, we try to replace all parameter uses of the method call with alternatives in the test that satisfy the above listed conditions, or else recursively delete the dependent methods as well.

When an individual is selected for mutation during the search, then each method call in the test case of length l is changed or deleted with probability $1/l$, and insertion happens with a predefined probability.

D. Crossover

Crossover of two method sequences is problematic as the parameter dependencies of a part of one of the parents might not be satisfied in the other parent. Consequently, we use the following crossover function: We select a crossover point p, and then for the first offspring append one statement after the other of the second half of one parent to the first half of the other parent. This insertion works same as the insertion used for mutation.

V. EVALUATION

To evaluate the approach presented in this paper, we conducted a set of experiments on the open source library

Table I
SOURCES FOR USAGE INFORMATION: NUMBER OF JODA-TIME CLASSES FOR WHICH INFORMATION EXISTS, NUMBER OF MODELS OF THESE CLASSES, AND CONSTRAINTS RESULTING FROM THE MODELS.

Name	Classes	Models	Constraints
Joda-Time Source	221	11198	11526
Joda-Time Tests	376	32865	33419
Σ	521	44653	45413

Joda-Time[1]. As source for code examples we used the Joda-Time library itself, and consider the set of JUnit test cases part of the Joda-Time distribution as client code that uses the API we are testing. Table I summarizes statistics on the usage information contained in these model sources. The number of classes listed in Table I includes member classes, anonymous classes, and classes belonging to the Joda-Time test suite, as these classes are contained in the same package.

For experimentation, we selected the top-level, concrete classes in Joda-Time, as these represent the API that will be most commonly accessed from the outside, resulting in a set of 26 classes. For each of these classes we ran test case generation without usage models, and using the usage models derived from the source code, the client code (tests), and the combined usage model, varying the probability of uncommon transitions between 0–100%. Resulting test suites are minimized after test case generation by successively removing statements until all remaining statements contribute to the coverage goal. For each configuration we ran 40 experiments with different random seeds, averaging the results.

Our prototype tool is implemented in Java, and implements a steady state genetic algorithm. For each run in the experiments, the search limit was set to 500,000 executed statements.

A. *Influence of Model Quality*

By construction, test cases generated from usage models follow the observed usage information. The resulting code coverage therefore depends on the *code examples from which common usage is learned.* To see how this influences the quality of resulting test suites in terms of code coverage, we measured the branch coverage of the existing, manually written test cases, and generated test suites without usage information and with each of the model sets. For each class and usage model, we generated 40 test suites using different random seeds.

Figure 7 shows the branch coverage averaged over all runs and all classes, comparing it to test cases generated using the same test generation techniques but without usage information. "Manual Tests" denotes the test cases that are included in the Joda-Time distribution. The information

[1]http://joda-time.sourceforge.net/

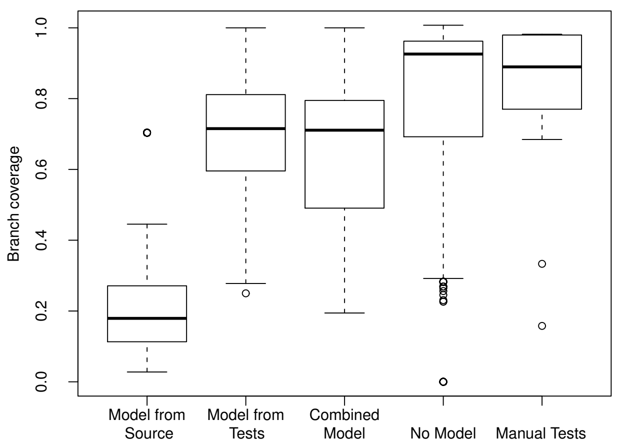

Figure 7. The branch coverage achieved using different model sources, averaged over 40 runs.

mined from the source code leads to relatively low coverage in the tests, which is in contrast to the large amount of models mined from the source code. We expect that this is because these models largely reflect usage of the internal API, and not the external API for which we are generating the tests.

The usage information contained in the test cases proved to be a good source of information. There is no coverage increase by combining the models, which is likely because the test case generation is misguided by incomplete information learned from the source code. Still, the coverage is about 10% lower than that of the manual test suite, which indicates room for improvement.

Finally, note that the coverage achieved without usage information ("no model") is higher even than the coverage of the manual test suite. However, as we will see in Section V-D, this additional coverage includes runs that are likely candidates for violating implicit preconditions.

The larger the variety in usage examples,
the higher the resulting coverage.

B. *Effect of Usage Models on Test Suites*

Figure 8 shows the number of test cases produced per class on average. The test suites derived from the client code model are overall the smallest, which is also due to the lower coverage. Interestingly, there is a huge difference in the number of test cases in automatically generated test suites vs. manually written test cases, even if the coverage increase is not overly large over automatically generated test cases. This can be interpreted as *redundancy* in the manually written test cases; it may show that it takes *extra effort* to

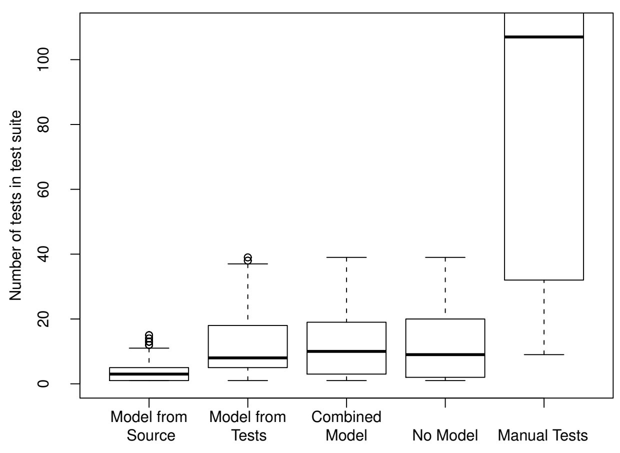

Figure 8. The number of test cases contained in a test suite using different model sources, averaged over 40 runs. The chart is cut off at size 110 to increase readability.

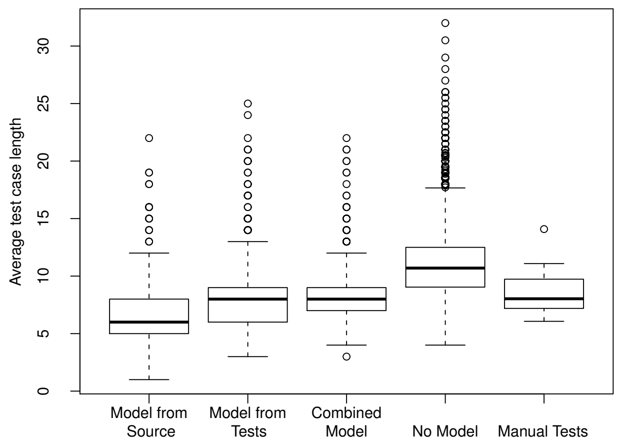

Figure 9. The average length per test case created using different model sources, averaged over 40 runs.

obtain the highest coverage; but it may also show that branch coverage is *not a good estimate* to measure test quality. Finally, Figure 9 shows that the average length of test cases created with usage information is significantly smaller than when using no usage information, which proves the expected reduction of “noise” (i.e., unneeded functionality).

Test cases derived from common usage are shorter than test cases generated without usage information.

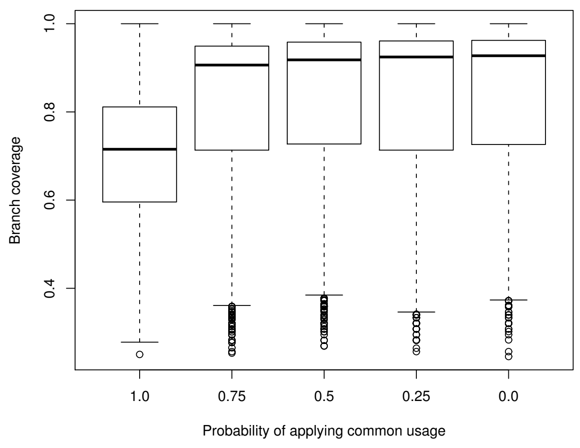

Figure 10. Box plot of coverage for different probabilities of random choice over common usage, varying from 1.0 (= only common usage) to 0.0 (= only random choice), for the test suites generated with the test usage model.

C. Exploration versus Common Usage

As seen in Figure 7, restricting exploration to already seen usage may reduce the resulting code coverage, depending on the amount of available usage information. Furthermore, in practice there can be scenarios where there is no usage information available for some of the classes. For example, when creating test cases for a newly written class which is not used anywhere in source code there exist no examples to learn from. Usage information can still be useful in order to create suitable parameter objects.

In practice, this means that there is a *trade-off* between *readability* and *coverage*: One will want to maintain as much readability as possible, but needs to allow exploration of new sequences to cover new parts of the source code. (Some of the high coverage may be explained by explicitly covering exceptional behavior, though, as we will discuss in Section V-E.) Our approach allows the tester to easily regulate between readability and exploration, by choosing the probability with which a transition is chosen out of the API Usage Model or randomly out of the entire set of possible methods.

Figure 10 illustrates the effects of varying the probability of random choice. It is interesting to see that once allowing random choice, the actual value of the probability has little influence on the achieved result in terms of coverage, given enough time for test case generation. In practice, these effects imply a simple *ranking:* testers can focus on common usage first, and explore uncommon usage until the test cases become too hard to understand.

By defining the amount of non-common usage, testers can seamlessly choose between readability and exploration.

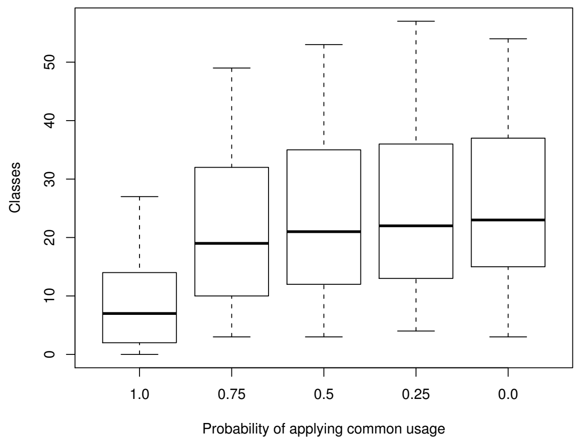

Figure 11. Diversity of classes used in the test cases for different probabilities of random choice over common usage, varying from 1.0 (= only common usage) to 0.0 (= only random choice), for the test suites generated with the test usage model. With increasing probability of uncommon transitions, the amount of "noise" (i.e., unneeded functionality) in the test cases increases.

D. Quantifying Readability

As stated in the introduction, our goal is to make test cases easier to understand. But how does one quantify readability? In Figures 1 and 2, we already have seen that readability can be associated with *the amount of functionality a test case relies upon*—the more functionality a test case needs to satisfy its purpose, the longer it takes to understand it. More specifically, we examine the *diversity* in terms of classes —the more classes a test case requires, the harder it is to understand.

In Figure 9, we already have seen that test case generation without usage information leads to longer test cases. To quantify the amount of diversity, we repeat the sensitivity analysis of Section V-C and count the different classes accessed in the test cases. Figure 11 shows that the number of different classes increases on average with increasing use of uncommon behavior. Interestingly, with *only* common usage allowed (probability 1.0), the number of classes is reduced by *68.1%* in contrast to test cases generated without usage information. (Keep in mind, though, that this increase in readability comes with a decrease in coverage, as shown in Section V-A).

In our experiment, test cases based on common usage use 68% fewer different classes, making them easier to understand.

E. Exceptional Behavior

After discussing the basic metrics of our test cases, let us now focus on their *effects*—specifically, the outcome of

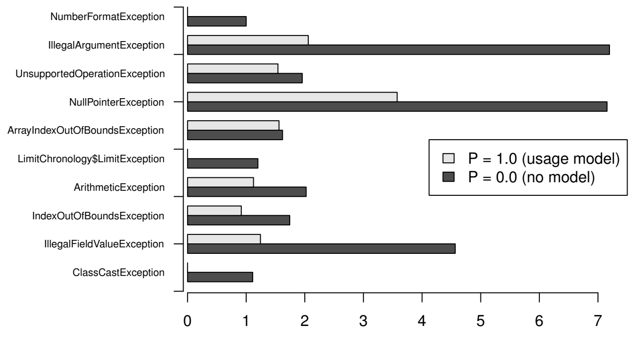

Figure 12. Number of exceptions raised per test suite per class, averaged over all runs on all classes.

test cases. In the absence of generated oracles, we can still check whether a test case raised an *exception*, and if so, of which kind. Figure 12 shows the number and distribution of exceptions for the tests generated with ($p = 1.0$) and without ($p = 0.0$) common usage model.

The striking observation is that with leveraging common usage, the number of exceptions is much lower. Why is this the case? As seen in Section V-A, test suites without model are longer, thus increasing the chance of triggering exceptions. But even if we normalize the number of exceptions per statement, on average we obtain 0.07 exceptions per statement with common usage model, in contrast to 0.14 without—that is, 50% less.

Manual investigation of this effect reveals that Joda-Time comes with a large number of built-in runtime checks, checking against illegal arguments. Not knowing whether these are part of the specification or not, our "no model" test case generator attempts to cover all these branches by specifically generating illegal arguments. Generating test cases based on common usage reduces the number of these exceptions, because common usage does not expect runtime exceptions. In practice, this again means that testers can focus on common usage first, and exceptional behavior later.

Test cases based on common usage raise fewer exceptions.

F. Implicit Preconditions

In testing practice, any unexpected exception raised by a test indicates an error—either in the code, or in the test. In the case of Joda-Time, we already know that the Joda-Time code already passes all the tests of the handwritten Joda-Time test suite (which is of high quality—see the above experiments for its coverage and extent). Therefore, we assume that any raised exceptions are there on purpose. If we had no confidence in the test subject, however, we would now be faced with assessing hundreds of test outcomes—out of which almost all would eventually turn out as false positives.

As stated earlier, some of the exceptions are raised intentionally by Joda-Time checks, possibly making them part of the specification—and thus imposing a test obligation. There are exceptions, though, that are never raised on purpose by Joda-Time, such as `NullPointerException`. Again assuming the high quality of Joda-Time, we can derive that any `NullPointerException` raised is an *involuntary effect of the test suite violating some implicit precondition*—that is, a nonsensical test case. Considering Figure 12 again, we see that `NullPointerExceptions` are far more prevalent in "no model" test cases. Normalized per statement, we find that a test case reflecting common usage only raises 48% as many `NullPointerExceptions` as a "no model" test case, reducing the number of nonsensical test cases by the same amount.

Test cases based on common usage have fewer violations of implicit preconditions.

VI. Threats to Validity

The results of our experiments are subject to the following threats to validity:

- **Threats to *external validity*** concern our ability to generalize the results of our study, and are common for any empirical analysis. In particular, as we only considered classes of a single open source library, we cannot claim that the results of our experimental evaluation are generalizable. Hence, the evaluation should be seen as investigating the potential of the technique rather than providing a statement of general effectiveness.
  Another threat to validity comes from the fact that we used the Joda-Time test suite to learn models from rather than actual client code. This is motivated by the fact that we did not find sufficient suitable Joda-Time clients to learn from. As demonstrated in Section V-A, our approach is clearly dependent on the quality of models; a sufficient number of client (or test) code is therefore required to make it work.
- **Threats to *internal validity*** concern our ability to draw conclusions about the connections between our independent and dependent variables. To reduce the probability of having faults in our framework, it has been carefully tested. As the implemented algorithms are randomized, we ran each experiment 40 times.
- **Threats to *construct validity*** concern the appropriateness of our measures for capturing our dependent variables. We measured the effects of our approach in terms of size, length, coverage, diversity, and violations of implicit preconditions. The measure of readability is only assessed indirectly; measuring the true effort for understanding would require human studies and is part of our future work.

VII. Conclusions

Automated code-based test generation techniques suffer from the problem that a tester needs to *understand* the tests in order to generate effective *oracles* and to weed out nonsensical test cases that violate implicit preconditions. In this paper, we have shown how to exploit the information contained in existing code examples in order to produce test cases that resemble real code. The resulting test cases are shorter, reference fewer different classes, and violate fewer preconditions, making them altogether more understandable and more valuable. These improvements come at the cost of achieving a lower coverage; however, the tester can seamlessly choose between readability and exploration and focus on common usage before exceptional usage. Even in the case of a newly written class for which no code examples exist, this approach can be useful in order to generate valid and understandable test inputs and parameters of complex data types.

There are several ways in which our work can be extended. Technically, our current prototype tool does not yet make full use of class hierarchies. For example, if class B inherits from class A, and we mine usage of B for a method that declares a parameter of type A, then this information will be used during test generation. However, if we want to test A directly, our prototype does not yet make use of information mined for its subclass B. Similarly, usage information related to class casts is not yet considered.

The usage models we used in this paper only analyze the interactions between classes. However, when re-visiting the test case example from Figure 2 in the introduction, another thing that can influence readability is the choice of primitive values. For example, it is not clear whether 197 and -91 denote years, milliseconds, or something else. Although we collect all constants that exist in the source code and reuse them during test case generation, it would require *dynamic* usage mining at *runtime* to collect more typical values for parameters of primitive datatypes. We expect that these improvements will lead to an increase in coverage.

In this paper we showed that learning and applying common object usage is feasible, and has an impact on the test cases. There is, however, no objective measurement for *readability* of test cases to date. Similarly, it is not possible to directly measure violations of implicit preconditions as, alas, these preconditions are not explicit. Future work will include work on quantifying readability, and setting up benchmarks that allow evaluation of test generation techniques with respect to how they treat implicit preconditions.

Acknowledgments. Gordon Fraser is funded by the Cluster of Excellence on Multimodal Computing and Interaction at Saarland University, Germany. We thank Andrzej Wasylkowski for his help in working with JADET, and Kevin Streit and Valentin Dallmeier for comments on earlier versions of this paper.

References

[1] J. A. Whittaker and M. G. Thomason, “A markov chain model for statistical software testing,” *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 812–824, 1994.

[2] G. H. Walton, J. H. Poore, and C. J. Trammell, “Statistical testing of software based on a usage model,” *Software: Practice and Experience*, vol. 25, no. 1, pp. 97–108, 1995.

[3] L. Llana-Díaz, M. Núñez, and I. Rodríguez, “Customized testing for probabilistic systems,” in *Testing of Communicating Systems*, ser. Lecture Notes in Computer Science, M. Uyar, A. Duale, and M. Fecko, Eds., vol. 3964. Springer Berlin / Heidelberg, 2006, pp. 87–102.

[4] P. A. Brooks and A. M. Memon, “Automated GUI testing guided by usage profiles,” in *ASE ’07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2007, pp. 333–342.

[5] P. Tonella and F. Ricca, “Dynamic model extraction and statistical analysis of web applications,” in *WSE ’02: Proceedings of the Fourth International Workshop on Web Site Evolution*. IEEE Computer Society, 2002, p. 43.

[6] S. J. Prowell, “TML: A description language for markov chain usage models,” *Information and Software Technology*, vol. 42, no. 12, pp. 835 – 844, 2000.

[7] ——, “JUMBL: A tool for model-based statistical testing,” in *HICSS ’03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS’03) - Track 9*. IEEE Computer Society, 2003, p. 337.3.

[8] G. H. Walton and J. H. Poore, “Generating transition probabilities to support model-based software testing,” *Software: Practice and Experience*, vol. 30, no. 10, pp. 1095–1106, 2000.

[9] W. Dulz, R. German, S. Holpp, and H. Götz, “Calculating the usage probabilities of statistical usage models by constraints optimization,” in *AST ’10: Proceedings of the 5th Workshop on Automation of Software Test*. ACM, 2010, pp. 127–134.

[10] J. Hao and E. Mendes, “Usage-based statistical testing of web applications,” in *ICWE ’06: Proceedings of the 6th International Conference on Web Engineering*. ACM, 2006, pp. 17–24.

[11] K. Sayre and J. Poore, “Automated testing of generic computational science libraries,” *Hawaii International Conference on System Sciences*, vol. 0, p. 277c, 2007.

[12] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, “MSeqGen: object-oriented unit-test generation via mining source code,” in *ESEC/FSE ’09: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2009, pp. 193–202.

[13] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth, “Dygen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces,” in *Tests and Proofs*, ser. Lecture Notes in Computer Science, G. Fraser and A. Gargantini, Eds. Springer Berlin / Heidelberg, 2010, vol. 6143, pp. 77–93.

[14] P. Godefroid, N. Klarlund, and K. Sen, “DART: directed automated random testing,” in *PLDI’05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2005, pp. 213–223.

[15] T. Xie, D. Marinov, W. Schulte, and D. Notkin, “Symstra: A framework for generating object-oriented unit tests using symbolic execution,” in *TACAS’05: The 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005, pp. 365–381.

[16] P. McMinn, “Search-based software test data generation: A survey,” *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[17] C. Csallner and Y. Smaragdakis, “JCrasher: an automatic robustness tester for Java,” *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[18] C. Pacheco and M. D. Ernst, “Randoop: feedback-directed random testing for Java,” in *OOPSLA’07: Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Application*. ACM, 2007, pp. 815–816.

[19] P. Tonella, “Evolutionary testing of classes,” in *ISSTA’04: Proceedings of the ACM International Symposium on Software Testing and Analysis*. ACM, 2004, pp. 119–128.

[20] S. Wappler and F. Lammermann, “Using evolutionary algorithms for the unit testing of object-oriented software,” in *GECCO’05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*. ACM, 2005, pp. 1053–1060.

[21] J. C. B. Ribeiro, “Search-based test case generation for object-oriented Java software using strongly-typed genetic programming,” in *GECCO’08: Proceedings of the 2008 GECCO Conference Companion on Genetic and Evolutionary Computation*. ACM, 2008, pp. 1819–1822.

[22] G. Fraser and A. Zeller, “Mutation-driven generation of unit tests and oracles,” in *ISSTA’10: Proceedings of the ACM International Symposium on Software Testing and Analysis*. ACM, 2010, pp. 147–158.

[23] K. Doerner and W. J. Gutjahr, “Extracting test sequences from a markov software usage model by ACO,” in *Proceedings of the 2003 International Conference on Genetic and Evolutionary Computation*, ser. GECCO’03. Springer-Verlag, 2003, pp. 2465–2476.

[24] A. Wasylkowski, A. Zeller, and C. Lindig, “Detecting object usage anomalies,” in *ESEC/FSE 2007: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2007, pp. 35–44.

[25] A. Arcuri and X. Yao, “Search based software testing of object-oriented containers,” *Information Sciences*, vol. 178, no. 15, pp. 3075–3095, 2008.