Automatically Generating Test Templates from Test Names

Benwen Zhang
University of Delaware
Newark, DE, USA
benwen@udel.edu

Emily Hill
Drew University
Madison, NJ, USA
emhill@drew.edu

James Clause
University of Delaware
Newark, DE, USA
clause@udel.edu

***Abstract*—Existing specification-based testing techniques require specifications that either do not exist or are too difficult to create. As a result, they often fall short of their goal of helping developers test expected behaviors. In this paper we present a novel, natural language-based approach that exploits the descriptive nature of test names to generate test templates. Similar to how modern IDEs simplify development by providing templates for common constructs such as loops, test templates can save time and lower the cognitive barrier for writing tests. The results of our evaluation show that the approach is feasible: despite the difficulty of the task, when test names contain a sufficient amount of information, the approach's accuracy is over 80% when parsing the relevant information from the test name and generating the template.**

I. Introduction

Software testing is an expensive and laborious activity that can account for the majority of the total cost of developing software. In the future, such high costs are likely to persist or even increase as the growing size and complexity of modern software exacerbates existing challenges. Techniques that improve the efficiency and effectiveness of the testing process can thus significantly reduce the overall cost of software development as well as improve software quality.

Fortunately, the software engineering research community has provided many techniques that can help developers reduce the costs of testing. One group of techniques attempts to completely remove the burden of testing by automating the test generation process (e.g., [13, 15]). While the tests generated by such automated techniques can be successful at revealing some types of unexpected behavior (i.e., sad paths), they are much less successful at helping developers test expected behaviors (i.e., happy paths). In addition, the tests often look very different from manually written tests and are difficult to maintain and understand [15].

In contrast to techniques that attempt to completely automate the test generation process, specification-based test generation techniques attempt to help developers test expected behaviors by creating tests from specifications (e.g., [5, 11, 12, 16, 20]). Ideally such specification-based approaches can decrease the costs of testing by eliminating the tedious and error-prone work of manually translating specifications into executable tests. However, they often fall short of this goal because the required specifications either do not exist or must be written in a format that takes as much, if not more, effort than manually writing the tests.

In this paper we present a new, specification-based approach that can help reduce the costs of testing by eliminating some of the tedious and error-prone work of writing unit tests. At a high-level, the approach leverages test names, which capture a developer's testing intent, to automatically generate test templates. Similar to how modern IDEs provide templates for common constructs such loops, try-catch blocks, method calls, etc., our approach infers what and how the developer wants to test and generates the corresponding code. Note that like the templates generated by IDEs, the test templates generated by our approach will likely have "holes"—locations where developers will need to provide additional information that is not included in the test name. Developers can fill in such holes by either providing the necessary information directly (e.g., adding concrete inputs for method calls) or by improving the test's name. Encouraging developers to write better test names in this manner is an additional benefit of the approach, potentially simplifying future maintenance tasks.

To assess the feasibility of the approach, we conducted a preliminary evaluation by manually comparing the templates generated by our approach to the original, developer written test bodies. Despite the difficulty of the task, when test names contain a sufficient amount of information, the approach's accuracy at parsing the relevant information from the test name and generating the corresponding template is over 80 %.

II. Motivation

Our choice to infer developer intent from test names is based on a combination of two observations. The first observation is that **every unit test must have a corresponding name**. While this may seem unremarkable, it does mean that developers are accustomed to providing such information. Therefore, using it as the basis for our approach means that, unlike for other specification-based techniques, there is no additional overhead for developers in either creating specifications or modifying them to fit a specific format. The second observation is that **test names should describe and summarize important parts of the test's body**. More specifically, they should indicate the scenario being tested and the expected outcome. As such, they can serve as a (partial) specification for the test.

As examples of the descriptiveness of test names, consider the tests shown in Figure 1. Despite coming from different projects and being written by different developers, the name of each test summarizes its body: in `testMaximumSize_negative`, the test sets the maximum size

```
public void testMaximumSize_negative() {
  CacheBuilder<> builder = new CacheBuilder<Object, Object>();
  try {
    builder.maximumSize(-1);
    fail();
  } catch (IllegalArgumentException expected) {}
}
```

(a) Guava test for the CacheBuilder class.

```
public void testEntrySetClearChangesMap() {
  Map map = makeFullMap();
  Set entrySet = map.entrySet();
  assertTrue(map.size() > 0);
  assertTrue(entrySet.size() > 0);
  entrySet.clear();
  assertTrue(map.size() == 0);
  assertTrue(entrySet.size() == 0);
}
```

(b) Apache Commons Collections test for the Map class.

```
public void testSettingHeightThatIsTooSmallLeavesHeightUnchanged() {
  Barcode barcode = new BarcodeMock("12345");
  int height = barcode.getHeight();
  barcode.setBarHeight(0);
  assertEquals(height, barcode.getHeight());
}
```

(c) Barbecue test for the Barcode class.

Fig. 1. Unit tests illustrating the descriptive nature of test names.

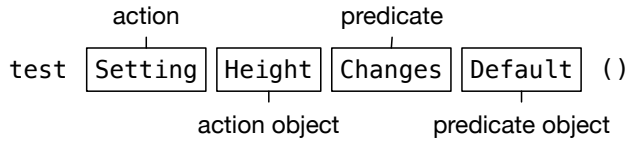

Fig. 2. Test name annotated with its parts of test.

of the cache builder to a negative number; in `testEntrySetClearChangesMap`, the test checks whether clearing the map's entry set changes the contents of the map; and in `testSettingHeightThatIsTooSmallLeavesHeightUnchanged()`, the test checks whether setting the barcode's bar height to zero changes its height.

III. Parts of Test

Beyond simply summarizing test bodies as shown in Section II, we have found that test names frequently follow a set of well defined grammatical rules. While in theory, developers have the full range of flexibility allowed by the programming language when naming tests, in practice, **test names often follow a relatively well defined grammatical structure**. This regular set of grammatical conventions makes it possible to extract relevant information from test names that relates to the test's body. More specifically, it allows for automatically tagging the name with what we refer to as parts of test (POTs)—the testing equivalent to parts of speech.

At a high-level, test names typically consist of two verb phrases: an *action phrase* and a *predicate phrase*. The action phrase describes the scenario being tested, including actions which should be performed and the state of the environment, whereas the predicate phrase describes the expected outcome of the test. For example, consider the test name shown in Figure 2. Here the action under test is "setting the height", and the result of that action is that "default" should be "changed".

In some cases, the action phrase and predicate phrase can be further broken down into individual components of a verb—an action or a predicate—and an object noun phrase—action object or predicate object. In Figure 2, the action is "setting", its object is "height" (what is being set), the predicate is "changes" and the predicate object is "default".

```
public void testGetDataReturnsData() {
  String data = ...;
  Barcode barcode = new Barcode(data);
  String actual = barcode.getData();
  assertEquals(data, actual);
}
```

(a)

```
public void testGetDataReturnsData() {
  String data = "12345";
  Barcode barcode = new BarcodeMock(data);
  assertEquals(data, barcode.getData());
}
```

(b)

Fig. 3. Comparison between the the test template generated by our approach (a) and the developer-written test (b).

Some very descriptive test names include additional information beyond the action or predicate phrase. For example, in `testSettingHeightThatIsTooSmallLeavesHeightUnchanged()`, the phrase "that is too small" describes how the action object (height) should be set. Similarly, in `testCopy_byteArrayToWriterWithEncoding()`, the phrases "to writer" and "with encoding" further describe how the byte array should be copied. These phrases are not part of the action phrase or predicate phrase, but rather modify, or further describe them. In our experience, we have observed that these modifiers describe the action phrase, and thus we refer to them as *action modifiers*.

In summary, test names typically contain one or more of the following POTs: action, action object, action modifiers, predicate, and predicate object.

IV. Automatically Generating Test Templates

Given a test name and a class under test, the approach creates a corresponding test template using two main phases. The first phase, *parsing*, involves parsing the test name to identify the relevant information it contains (i.e., the action phrase and the predicate phrase). Note that, because of potential ambiguities in how a test name can be interpreted, multiple parses may be produced by this step. The second phase, *generation*, takes as input the set of parses produced by the first phase, and the class under test, to produce one or more test templates for each parse. To produce the test templates, the generation phase statically analyzes the code of the application under test to map the identified POTs to methods, fields, parameters, assertions, etc. Again, multiple templates may be generated for a parse because of potential ambiguities in how the POTs can be mapped to executable code. For example, there may be multiple methods with the same name or multiple ways to access a field. In practice, developers can choose among the generated test templates in the same manner they currently choose among possible autocompletion options, with the IDE allowing them to quickly cycle though the options.

A. Concrete Example

As a concrete example of the output of each phase, assume that a developer wants to create a test named `testGetData-`

`ReturnsData` to test the Barbecue application's Barcode class. The first phase analyzes the name and tags "get" as the *action*, "data" as the *action object*, "returns" as the *predicate*, and "data" as the *predicate object*. Given this parse, the second phase generates the template shown in Figure 3(a). As a point of comparison, the developer-written test body for this test is shown in Figure 3(b).

In the test template, an instance of Barcode, the class under test, is instantiated. Note that the constructor requires a single argument of type `String`. Because the test name does not include any information about how the environment should be setup (i.e., any action modifiers), a hole is left by the approach, but the necessary type is clearly indicated. Note that in this case, the approach could easily provide a randomly selected string as the specific value is irrelevant. However, because this fact is difficult to infer in general, the approach leaves the hole for the developer to complete instead.

To generate the scenario under test part of the template, the approach examines the code of Barbecue in order to map the action phrase, "get data", to executable code. Here, the approach finds a method of the Barcode class called `getData` that matches the action phrase. To generate the expected outcome part of the test, the approach matches the predicate, "returns", to a predefined assertion method. In this case, the most appropriate assertion is `assertEquals` because the intention is to compare the result of performing the action. Finally the predicate object, "data", is added to the predicate and compared against the result of getting the barcode's data. While it may be surprising that the approach can identify that the string argument provided to the constructor should also serve as the expected value in the assertion, it makes this connection using static analysis. Analyzing Barbecue reveals that the constructor's parameter is named "data". With this additional information, it becomes possible to infer that the data returned by `getData` should be the same data that was used to create the barcode.

The fact that the test template generated by our approach closely matches the developer-written test body demonstrates the usefulness of our approach. Instead of having to type the name and then repetitively type the same information in the body, the developer can rely on the approach to generate an appropriate template. Beyond the test name, the developer only needed to provide a concrete value for the data used to create the barcode. By eliminating the need to type test bodies, the approach allows developers to focus on choosing relevant test data or simply move on to other tasks more quickly.

B. Phase 1: Parsing Test Names

The goal of the parsing phase is to identify the action phrase and predicate phrase contained in a test name, when these phrases exist. Our approach for identifying such phrases is the result of combining information about English grammar, knowledge of (and intuition about) the conventions developers follow when writing tests, and a manual examination of existing test cases.

1) Extracting Grammatical Relations: The first step in identifying action phrases and predicate phrases is to use an off-the-shelf parsing system for English to identify the grammatical relations among the words in the test name (excluding the leading "test"). The result of this step is a semantic graph that shows the grammatical relationships among the words and their part of speech (POS). All of the semantic graphs extracted directly from the sentence fragment are passed onto the next step to identify the POTs.

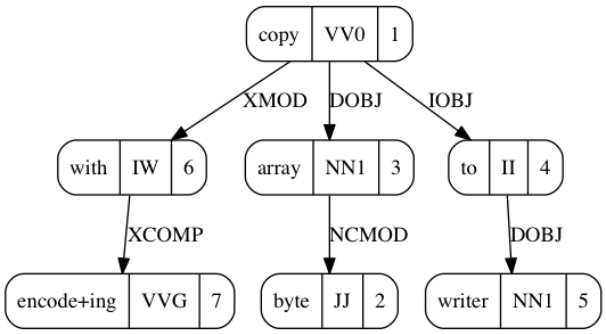

Fig. 4. Semantic graph for `testCopy_byteArrayToWriterWithEncoding()`.

Figure 4 shows one of the semantic graphs that results from parsing `testCopy_byteArrayToWriterWithEncoding()`. Each node in the graph represents an occurrence of a *word* in the sentence fragment and its *part of speech* based on the CLAWS7 Tagset (verb, noun, preposition, etc.). Common word stems are indicated by "+". For instance, "encoding" is an "-ing" verb participle (VVG), and it has the stem "-ing". The edges in the graph represent grammatical relations such as subjects (XSUBJ, NCSUBJ), clausal compliments when overt subjects are missing (XCOMP, CCOMP), clausal modifiers (XMOD, NCMOD, CMOD), as well as direct and indirect objects (DOBJ, IOBJ, OBJ2).

2) Identifying parts of test: The second step in identifying action phrases and predicate phrases is to use the semantic graph to tag the test name with its POTs. In identifying POTs, we are looking to identify the action and predicate phrases as well as action modifiers. Recall that simple sentences in English consist of a subject, verb, and then an object. (Note the use of the word "subject" in this subsection is distinct from the test's subject, or class under test, used elsewhere in the paper.) If the semantic graph includes a word with the role of subject, then the approach identifies that subject as the action object, its verb as the predicate, and the verb's corresponding direct or indirect object as the predicate object. Prepositions other than "to" denote action modifiers. If the semantic graph does not contain a subject and the test name begins with a verb, we assume there is no predicate in the name and attempt to extract only an action phrase and action modifiers.

One of the major challenges in identifying POTs is robustness. A test name may yield many possible semantic graphs, some of which may be perfectly correct, some that are workable, and some that are too incomplete to analyze. Thus, we have designed our approach to be as robust as possible in the face of errors in the grammatical relation parsing and part of speech tagging. We take a best-effort approach that attempts to match as many rules as possible and prioritizes the output of our approach by the most frequently identified POTs.

C. Phase 2: Generating Test Templates

The goal of the generation phase is to transform the POTs provided by the parsing phase into test templates. At a high-level, the process for transforming a parse into a test template is done in two steps. The first step is to convert the action phrase into the scenario under test part of the template and the second step is to convert the predicate phrase into the expected outcome part of the template.

1) Generating the Scenario Under Test: Intuitively, action phrases are converted into the scenario under test part of a template by searching the code of the application under test in order to find sequences of method calls and field accesses that correspond to the elements of the action phrase. Briefly, the search process starts with the class under test and performs a breadth-first traversal along the field types and method return types of the encountered classes looking for entities that match the specified action. The path between the identified action and the subject is then the sequence of method invocations and field accesses that will be performed by the test. In order to facilitate different ways of matching, we use the strategy pattern to allow the search process to transparently use different matchers. After the search is complete the matched fields and methods are ranked according to the matcher that identified them.

2) Generating the Expected Outcome Checks: The process of converting the predicate phrase into the expected outcome part of the template is simpler than converting the action phrase into the scenario under test. Because there are only a small number of possible assertions, we do not need to use a full search process. Rather we can use a set of manually created rules that look for specific words in the predicate phrase as well as the attitude of the test name to choose an assertion. If negative words such as "not" or "cannot" occur anywhere in the test name, we classify the attitude as negative, otherwise the attitude is positive.

For example, if the predicate contains the words "true" or "false", we assume that the corresponding assertion should be `assertTrue` or `assertFalse`, respectively. If the predicate is a linking verb, the predicate object is a constant, and the attitude is positive, we choose "assertEquals". If the predicate contains the words "throw", "fail" or "exception", `fail` will be chosen. Currently, this manually generated rule system is adequate, but we plan to improve it in future work by using machine learning techniques to generalize these rules from a large set of actual test names and source code. Such information will allow for a more accurate mapping of predicate phrases to assertions.

The final step in the template generation process is to combine the scenario under test part of the template and the expected outcome part of the template. Unlike other test generation approaches, the templates generated by our approach do not need to be executable, which can simplify the process. Instead of needing to construct potentially complex data, we can simply leave a hole that the developer can customize to suit their particular testing goals.

We identified five common patterns for the test templates: the *do and affect pattern*, the *no effect pattern*, the *before and after pattern*, the *throw exception pattern* and the *default pattern*. Of the five patterns, the default pattern is the most common. It follows a simple sequential input and oracle structure that consists of initializing the subject class, a method sequence invocation, and asserting on the returned value. The do and affect pattern is similar, but slightly different in that it invokes a method sequence in the input part of the test and asserts the returned value from another invocation of a method sequence or field access. This pattern is especially suitable for testing paired getters and setters. The no effect pattern and the before and after pattern are similar in that the input and oracle parts of these two pattern are interlaced. Their code usually follows an order of initializing the subject class, a method sequence invocation, asserting a returned value or field, another method sequence invocation, and another assert on a returned value or field. The difference between these two patterns is their different testing objectives. The no effect pattern checks whether an action unexpectedly affects the status of the object under test. On the other hand, the before and after pattern tests whether the input changes the status of the object under test in expected way. The last pattern, the throw exception pattern, tests whether the input causes an exception to be thrown.

V. Evaluation

The goal of our evaluation is to determine whether our approach can automatically create test templates that closely match tests written by developers. To perform a meaningful evaluation with respect to this goal, we must therefore consider test names that are descriptive; if we considered test names that are void of meaning, it would be difficult to assess the benefit that our approach can provide. Unfortunately, not every test name is descriptive. In practice, because naming is difficult and there is no immediate downside to choosing poor names, developers often create generic test names (e.g., test1, test2, etc.) or names that contain very little information (e.g., testAdd, testSubtract, etc.).

Given a descriptive test name, we determine if the generated test template bears any resemblance to the test body written by the original developer. Since there may be many semantically equivalent code sequences that test the same functionality, our goal is not to generate the exact same test case as the original developer. As such, we determine feasibility of our approach by investigating whether the basic components (object under test, action method call, assert structure) match the developer-written test.

A. Test Name Descriptiveness

To avoid the potential bias that could result from choosing descriptive test names ourselves, we instead had Masters and PhD students from the University of Delaware's Software Testing and Maintenance class label test names with their POTs. In the remainder of this section, we refer to these graduate students as *labelers*.

First, we used a weighted random selection strategy to choose 100 test names from the test suites of Barbecue, Commons-beanutils-1.8.3, Commons-cli-1.2, Commons-collections-3.2.1, Commons-io-2.4, DataStructures and Gson-2.2.4. Because labeling test names with their POT is time consuming—labelers took from 45 min to 60 min to label the POTs in 15 names—and we want to choose as many descriptive test names as possible, we biased the selection by the number of words in each test name; presumably test

```
public void testIOExceptionStringThrowable() {
  Throwable cause = new IllegalArgumentException("cause");
  IOExceptionWithCause exception =
     new IOExceptionWithCause("message", cause);
  assertEquals("message", exception.getMessage());
  assertEquals(cause, exception.getCause());
  assertSame(cause, exception.getCause());
}
```

Fig. 5. Name containing an action phrase that does not describe the test.

names that contain more words are more likely to contain more information. To count the number of words contained in a test name, we stripped the leading "test" and used a purpose build identifier splitter. The final set contained 10 test names with 1 or 2 words, 30 test names with 3 words, and 60 test names with 4 or more words.

After choosing the set of 100 test names, we randomly assigned 15 names to each labeler. Because we had 20 labelers, assigning 15 names to each allowed us to keep the expected length of the task below 1 h and to have each test name labeled three times. For each test name, the labeler was shown the test name as well as the name of the class under test and a brief description of the project from which the test name was taken. The labeler was then asked to identify which part of the test name should be tagged with each of the five POTs: action, action object, action modifiers, predicate, and predicate object. If a labeler felt that the test name did not contain a specific POT, they were instructed to enter a special "N/A" value. Due to labeler attrition, 10 of the test names were not labeled a sufficient number of times and were removed from the set leaving us with 90 labeled test names.

We then calculated how often a majority of labelers agree on whether a test name contains each POT: for action, labellers agreed 73 % of the time; for action object, the labellers agreed 68 % of the time; for action modifiers, the labellers agreed 49 % of the time; for predicate, the labellers agreed 80 % of the time; and for predicate object, the labellers agreed 75 % of the time. As this data shows, tagging test names with their POT is a relatively difficult task.

Using the POTs assigned by the labelers and the agreement scores, we identified which test names contain a sufficient amount of information as follows: test names where a majority of labelers agree that it contains either (1) an explicit action (i.e., not "get" or "set"), or (2) an implicit action (i.e., "get" or "set") and an action object are presumed to contain an action phrase; and test names where a majority of labelers agree that it contains both a predicate and a predicate object presumably contain a predicate phrase. Under this classification, 60 % of the test names (54 out of 90) have enough information for the template to include the scenario under test (i.e., they contain an action phrase) and 43 % (39 out of 90) have enough information for the template to include the expected outcome (i.e., they contain a predicate phrase).

B. Comparing templates with developer-written tests

After identifying which test names contain a suitable amount of information, we generated the corresponding test templates for the 54 test names that have an action phrase and the 39 test names that have a predicate phrase. We manually compared the templates against the developer-written test bodies to determine whether they match. We label a test template as a match if it invokes the exact same method as the main test action (which requires instantiating a similar type of object), and when it uses the same assertion pattern (including similarly typed actual and expected values). For templates that do not match the developer-written test body, we classified the cause of the mismatch as either an irrelevant name, an incorrect parse, or an incorrect template.

TABLE I. RESULTS OF COMPARING THE TEMPLATES GENERATED BY OUR APPROACH TO THE DEVELOPER-WRITTEN TEST BODIES.

Outcome	Action phrase	Predicate phrase
Match	31	29
Incorrect parse	7	4
Incorrect template	6	4

The first mismatch cause, *irrelevant name*, was assigned when the test name did not describe either the scenario under test or the expected outcome. Although our labelers agreed that these names contained an action phrase or a predicate phrase, the information from the name does not summarize the test body. Figure 5 shows an example of an irrelevant name. In this case, the action phrase—get IOException—does not describe the scenario being tested, creating a Throwable `IllegalArgumentException`. The second mismatch cause, *incorrect parse*, was assigned to when the parser incorrectly identified the information contained in the test name. The third mismatch cause, *incorrect template*, was assigned when the parser correctly identified the information contained in the name but the template generator chose an incorrect method, field sequence, assertion, etc.

Because cases where the name of the test is irrelevant are equivalent to cases where the test name does not contain sufficient information, we removed them from further consideration. If the specification is wrong, it is nearly impossible for the template to be correct. After removing the tests with irrelevant names, we were left with 44 relevant test names that contain an action phrase and 37 relevant test names that contain a predicate phrase.

Table I shows the results of comparing the test templates for the relevant test names to the developer-written test bodies. The first column, *Outcome*, shows whether the template matched, was a parser mistake, or template generation mistake. The second and third columns, *Action phrase* and *Predicate phrase*, show, for the test names that contain an action phrase or predicate phrase, respectively, the number of times each outcome occurred.

When considered individually, the accuracy of the components of the approach are above 80 %. More specifically, for test names that contain an action phrase, the parser was able to extract the correct information 82 % of the time ($\frac{31}{44-6}$) and the generator was able to generate a template that matched the developer-written body 84 % of the time ($\frac{31}{44-7}$). For test names that contain a predicate phrase, the parser was able to extract the correct information 88 % of the time ($\frac{29}{37-4}$) and the generator was able to generate a template that matched the developer-written body 88 % of the time ($\frac{29}{37-4}$). Based on this data, we conclude that it is feasible to automatically generate test templates based on the developer intent expressed in test names.

VI. Related Work

Specification-based Test Case Generation: At a high-level, specification-based approaches share our overall goal of reducing the cost of writing tests by automatically generating tests from some form of documentation. Specific types of documentation that have been considered include: System Description Languages (SDLs) [20], use cases [11], UML [12], and scenarios [16]. In addition to generating test cases, specification-based techniques have also been used to generate application code [5]. In contrast to the English test method names used by our approach, the forms of documentation required by other specification-based approaches are less likely to exist and are more difficult to create.

Code Completion: Code completion techniques attempt to reduce the costs of coding, rather than testing, by reducing a developer's searching and typing. Various code completion techniques have been developed based on Hidden Markov models (HMMs), ranking, statistical properties of source code, machine learning, data mining, as well as other matching approaches to recommend APIs according to users' incomplete inputs (e.g., [4, 6]).

Code Snippet Search: Code snippet search techniques attempt to locate and retrieve relevant source code in response to a query. Related work in this field include experiments to evaluate diverse approaches to code search [17]; techniques to improve the searchability of source code (e.g., [9, 10]); techniques for extracting information from diverse sources (e.g., [14, 19]); and interactive techniques (e.g., [1, 21]).

Natural Language Program Analysis: Finally, in the area of Natural Language Program Analysis (NLPA), researchers have investigated identifier splitting techniques (e.g., [2, 7]); the technique for tagging words with their part of speech and identifying large semantic structures [3]; the technique for identifying programming-specific synonyms and antonyms [8]; and the comment generation technique [18]. These techniques are not alternatives to our approach but can be used to improve its accuracy and effectiveness.

VII. Conclusions

We presented a novel specification-based approach for automatically generating test templates based on the developer intent expressed in test names. The approach can help reduce testing costs by automating some of the tedious and error-prone work searching for code and manually writing tests. In addition, it encourages developers to write more descriptive test names, which can provide long-term benefits by simplifying test comprehension and maintenance tasks. Our preliminary evaluation provides evidence that the approach is feasible and promising: despite the difficulty of the task, when test names contain a sufficient amount of information, its accuracy at parsing the relevant information from the test name and generating the corresponding template is over 80 %.

VIII. Acknowledgments

This work is supported in part by National Science Foundation Grant No. 1527093.

References

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

[2] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc. Tidier: An identifier splitting approach using speech recognition techniques. *Journal of Software: Evolution and Process*, 25:575–599, 2013.

[3] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 3–12, 2013.

[4] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 27–38, 2013.

[5] M. Hamri and G. Zacharewicz. Automatic generation of object-oriented code from DEVS graphical specifications. In *Proceedings of the Winter Simulation Conference*, pages 409:1–409:12, 2012.

[6] S. Han, D. R. Wallace, and R. C. Miller. Code completion from abbreviated input. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343, 2009.

[7] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker. An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19(6):1754–1780, 2014.

[8] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 377–386, 2013.

[9] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, pages 664–675, 2014.

[10] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120, 2011.

[11] C. Nebut, F. Fleurey, Y. Le Traon, and J. Jezequel. Requirements by contracts allow automated system testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 85–96, 2003.

[12] J. Offutt and A. Abdurazik. Generating tests from uml specifications. In *Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard*, pages 416–429, 1999.

[13] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, 2007.

[14] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 832–841, 2013.

[15] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 23–32, 2011.

[16] J. Ryser and M. Glinz. A practical approach to validating and testing software systems using scenarios. In *Proceedings of the Third International Software Quality Week Europe*, 1999.

[17] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes. How well do search engines support code retrieval on the web? *ACM Trans. Softw. Eng. Methodol.*, 21(1):4:1–4:25, 2011.

[18] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th IEEE International Conference on Automated Software Engineering*, 2010.

[19] S. Subramanian and R. Holmes. Making sense of online code snippets. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 85–88, 2013.

[20] L. Tahat, B. Vaysburg, B. Korel, and A. Bader. Requirement-based automated black-box test generation. In *Proceedings of the 25th Annual International Computer Software and Applications Conference*, pages 489–495, 2001.

[21] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213, 2007.