Test generation for GUI product lines

Svetoslav Ganov
Laboratory of Experimental Software Engineering, University of Texas at Austin
svetoslavganov@mail.utexas.edu

Abstract

A Graphical User Interface (GUI) is composed of a set of widgets. Since the state of a GUI is modified by events on the GUI widgets, a test input for a GUI is an event sequence. Due to the combinatorial nature of these sequences, testing a GUI thoroughly is problematic and time-consuming. The possible values for certain GUI widgets, such as a textbox, are also combinatorial compounding the problem.

Software product lines are families of programs developed as compositions of features – crosscutting concerns that encapsulate certain functionality. Due to the combinatorial nature of these feature compositions, developing a test suite for each member of the product line is ineffective and time-consuming.

This paper presents an approach for testing GUI product lines by reusing tests developed for individual product line members. We define a GUI test case as an event sequence, a set of data inputs, and a set of GUI states observed during the execution of the event sequence. Such an approach allows us to detect reusable tests. We utilize the reused tests as a seed for test generation.

1. Introduction

A Graphical User Interface (GUI) provides a convenient way to interact with the computer. GUIs consist of virtual objects (widgets) that are intuitive to use, for example buttons, edit boxes, etc. While they have become ubiquitous, testing them remains largely ad-hoc. In contrast to console applications where there is only one point of interaction (the command line), GUIs provide multiple points each of which might have different states.

A classic challenge in GUI testing is how to select a feasible number of event sequences, given the combinatorial explosion due to arbitrary event interleavings. Consider testing a GUI with five buttons, where any sequence of button clicks is a valid GUI input. Exhaustive testing requires trying all 120 possible combinations since triggering of one event before another may cause execution of different code segments.

An orthogonal challenge is how to select values for *data widgets*, i.e., GUI widgets that accept user input, such as textboxes, edit-boxes and combo-boxes, and can have an extremely large space of possible inputs. Consider testing a GUI with one text-box that takes a ten character string as an input. Exhaustive testing requires 1026 possible input strings (assuming we limit each character to be only alphabetical in lower-case).

A product line is a family of programs developed via composing a set of features and a base program. Each feature represents a crosscutting concern and encapsulates certain functionality. For example, a feature for a text editor could be the capability of undo and redo operations. Features consist of various artifacts (classes, class refinements, documents, document refinements, etc.) that represent transformations that must be applied to the program to introduce the new functionality.

A classic challenge is developing a test suite for each member of a program product line, given the combinatorial nature of feature compositions. Consider a product line generated by composing a set of five features, which may interact i.e., the composition order matters. Such a product line can be composed of up to 325 programs generated as permutations of up to five features. Testing this product line could potentially require 325 test suites.

An orthogonal challenge is how to automate test generation, given the potential feature interactions, where one feature may significantly modify the behavior of another feature and vice versa. Consider a base program that takes as an input an arbitrary string value, adding a validating feature that restricts the inputs to be numeric, renders all alphabetical values invalid. Testing the composed program requires either development of a new tests suite or adapting the test suite developed for the base program. Adapting an existing test suite may require adding new tests cases, discarding of existing ones, or adapting the existing tests to conform to the new input specification.

We present an approach for testing GUI product lines that addresses the combinatorial explosion of: (1) GUI event sequences and data inputs; (2) compositions of potentially interacting features. We reuse tests developed for individual members of the product line and utilize the reused tests as a seed for test generation. To enable detection of reusable tests, we define a GUI test case as an event sequence, a set of data inputs, and a set of GUI states observed during the execution of the event sequence and populating the input values.

We make the following contributions:

- **Reusable test detection.** We identify tests developed for individual members of product line that can be reused for testing other members this product line.
- **Novel test generation approach.** We utilize reusable tests as a seed for test generation. As a result we generate a test suite for each member of the product line.
- **Detection of feature interactions.** Our approach detects GUI feature interactions. We exploit the test reuse patterns to reason about these interactions.
- **Implementation.** Our framework, Barad, is fully automatic, performing test generation, test execution, and reusable test detection.
- **Evaluation.** We evaluate the applicability of our approach on a GUI product line. Our approach significantly reduces the time for generation of a test suite for the product line.

2. Example

In this section we provide an overview of our approach for testing of GUI product lines. Our goal is to give the reader intuition about our technique and demonstrate how it utilizes reuse of tests to generate test suites for individual members of the product line.

2.1. Simple Calculator

The GUI presented in Figure 1 is an application (202 lines of code) that we developed. It calculates the unit price rounded to the closest integer given a total amount and quantity. This version of the GUI represents the base program in its product line and performs a single operation, which is the core functionality for the programs in the product line. No validity checks are performed for the inputs in the total amount and quantity fields, which accept integer values. Hence, it is possible to enter an unrealistic negative value as a total amount or quantity. Note that the *Rounded unit price* text box is read only.

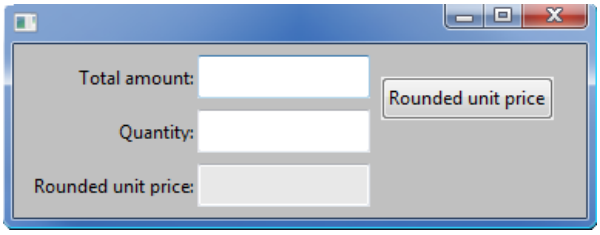

Figure 1. Screen shot of the *Simple Calculator*

2.1.1. Test generation. Since the *Simple Calculator* is the base program in its product line, there are no developed test cases for it. We use our GUI testing framework, Barad, to generate a test suite for the *Simple Calculator*. The test generation algorithm we use populates the values of *data widgets* – widgets that take data inputs from the user – adopting a specification based approach – selecting from a predefined set of values. For text field of the GUI we choose a value from the set {*-1*, *10*, the empty string}. Barad generates exhaustively event sequence up to a given bound. Each event in these sequences, transitions the GUI to a new state and no two identical states are observed during execution of the generated test suite (except the initial state).

Barad also supports replaying of tests on a member of the product line, which tests could have been generated for another member of that product line. During test execution Barad compares states stored in the test (observed during execution on the program the test was generated for) to the states observed during test execution on the currently tested program. In replay mode, Barad executes the replayed tests followed by test generation using as a seed for that test generation successfully executed (i.e. replayed) tests. Section 4 presents a detailed description of our approach.

We used Barad to generate and execute a test suite for the *Simple Calculator* and also replayed the generated test suite on the GUI. Table 1 shows the results.

Table 1. Test generation metrics for the *Simple Calculator*

Mode	Added	Pruned	Reused	Rejected	Time(s)
Generate	22	83	0	0	11.46
Replay	0	0	22	0	4.192

The first column contains the execution mode. Columns two, three, four, and five present the number of generated, pruned during the generation, reused during the replay, and rejected during the replay tests, respectively.

2.1.2. Result interpretation. As seen from Table 1, Barad generated and successfully replayed twenty two test. During the test generation, some test were pruned

(discarded) because they generate already observed program states.

Note that no tests were pruned during the replay, even though Barad uses the successfully replayed tests as a seed for test generation. During the replay Barad detected that the test suite is generated for the GUI on which it is currently executed. Since our framework generates test exhaustively, unnecessary test generation was avoided. Details about this process are presented in Section 4.

2.2. Persisting Calculator

The GUI presented on Figure 2 is the *Persisting Calculator*. It is another member of the product line and was generated by composing the *Simple Calculator* (i.e. the base program) with a persistence feature. The persistence feature mimics the functionality of saving the calculation result. This feature is composed of a single button, selecting of which performs a void action, simulating a non-interacting feature. Hence, the *Persisting Calculator* is a composition of the base program with an orthogonal feature.

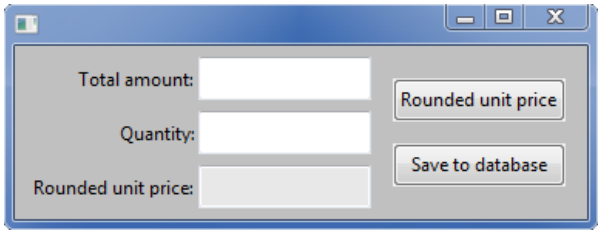

Figure 2. Screen shot of the *Persisting Calculator*

2.2.1. Test generation. The test generation approach, we used for the *Persisting Calculator*, differs from the one for the *Simple Calculator*. To evaluate the applicability of our approach for testing GUI product lines, we use the test suite generated for the *Simple Calculator* as a starting point for generating a test suite for the *Persisting Calculator* GUI. This way, we simulate a scenario, in which only tests for the base program are available and are used for generation of tests for other members of that product line.

We first generated a test suite for the *Persisting Calculator* without replaying the test suite generated for the *Simple Calculator*. Then we generated a test suite by replaying the test suite for the *Simple Calculator* and utilizing reused test as a seed for test generation. Note that during the test generation we used the same set of input values for populating the GUI data widgets. The obtained results are presented in Table 2, the structure of which is similar to that of Table 1.

Table 2. Test generation metrics for *Persisting Calculator*

Mode	Added	Pruned	Reused	Rejected	Time(s)
Generate	22	98	0	0	10.92
Replay	0	15	22	0	4.33

2.2.2. Result interpretation. As seen from the results in Table 1, the test generation without replay produced twenty two test cases and pruned ninety eight which generated duplicate states. During the replay phase all tests generated for the *Simple Calculator* were reused, which is due to the lack of interaction among the base program and the persistence feature. Since the test suite was generated for another member of the product line we utilize the reused tests to seed test generation. The lower number of pruned tests in the replay mode is due a heuristic that we use to prune regions of the event input space. Details about this heuristic are provided in Section 4. The fifteen pruned test cases are due to the increase in the event space, resulting from adding the *Save to database* button and the seeding with reused tests. Note that the persistence feature performs a void action, hence does not introduce new GUI states and no additional test cases were added to the test suite. Our results show that in this case our approach increased the speed of tests generation almost three times. Note that by reusing all test cases for the *Simple Calculator* we detected that the *Simple Calculator* feature (i.e. the base program) does not interact with the persistence feature.

2.3. Validating Calculator

The third member of the example product line is the *Validating Calculator*. We do not provide a screen shot of the application because it has the same structure as the *Simple Calculator* shown on Figure 1. The *Validating Calculator* is composed of the *Simple* the base program and a validation feature. As presented in Section 2.1 the *Simple Calculator* has two text boxes that accept any numeric value. The validation feature ensures that only non-negative values are populated in the *Total amount* and *Quantity* fields. Note that the validation feature interacts with the *Simple Calculator* by changing the valid inputs for its data widgets.

2.3.1. Test generation. The test generation approach for the *Validating Calculator* was similar to the one for *Persisting Calculator*. We first generated a test suite without replaying the tests for the *Simple Calculator* followed by replaying of those tests. During the test generation we used the same set of values for the data widgets as the values used for the other two members of the product line. The results are presented in Table 3, the structure of which is similar to that of Table 1.

Table 3. Test generation metrics for the *Validating Calculator*

Mode	Added	Pruned	Reused	Rejected	Time(s)
Generate	9	47	0	0	4.85
Replay	0	38	9	13	4.97

2.3.2. Result interpretation. As seen from the results in Table 3, nine tests were generated for the *Validation Calculator* without a replay. The test that generated duplicates states, were pruned. Note that using the replay mechanism we reused nine tests and rejected thirteen. The rejected tests generated GUI states that are not contained in the test cases for the *Simple Calculator*. By rejecting such tests we detected feature interaction i.e., the validation feature interacts with the base program. In such case we have to explore the entire event space (more precisely, GUI state space), since the feature interaction could change dramatically the application behavior. Hence, tests that generated duplicated states for the *Simple Calculator* may now generate different states for the *Validating Calculator*. As a consequence we measured almost the same execution time in generation and replay mode. Hence, in case of interaction features we were able to reuse test cases but did not gain decrease in the execution time. Note however, that we detected the feature interaction.

2.4. Discussion

The example product line demonstrates that our approach enables reuse of test cases generated for one member of a product line for testing other members of that product line. Further, our technique detects interacting features (see Section 4.4). In case of non interacting features we achieve significant increase in the speed of test generation. In case of interacting features we achieve the same speed of test generations as without reusing test cases. Hence, our approach opportunistically reuses test cases, increasing the test generation speed and in the worse case it has the same performance as complete generation.

3. Background

This section provides the reader with some background about the testing of software product lines. It also presents the traditional GUI testing approaches and the GUI model we adopted.

3.1. Software product lines

A software product line is a family of programs composed by a set of features. Various feature compositions enable fast prototyping, which enables effective development of user or market specific programs. The objectives are to maximize user satisfaction by providing highly customizable software products, speed up the development process, minimize development costs, increase productivity, and improve the quality of software.

3.2. GUI testing

Since contemporary software extensively uses GUIs to interact with users, verifying GUI's reliability becomes important. There are two approaches to building GUIs and these two approaches affect how testing can be done.

The first approach is to keep the GUI light weight and move computation into the background. In such cases the GUI could be considered as a "skin" for the software. Since the main portion of the application code is not in the GUI, it may be tested using conventional software testing techniques. However, such an approach places architectural limitations on system designers.

The second approach is to merge the GUI and its computations. The most common way of testing such GUIs is by using tools that record and replay event sequences [17]. This is laborious and time consuming. Another technique for checking GUI's correctness is by using tools for automatic test generation, execution, and assessment as the one presented in this paper or the ones described in [2] [4] .

3.3. GUI model

We take a standard view of each GUI. Let $W = \{w_1, w_2, ...w_n\}$ be the set of GUI widgets. Examples of widgets are *Button, Combo, Label*, etc. Each widget has a set of properties. Let $P = \{p_1, p_2, ...p_m\}$ be the set of widget properties. Examples of properties are *enabled, text, visible, selection*, etc. Each property has a set of values. Let $V = \{v_1, v_2, ...v_p\}$ be the set of property values. Examples of values are *true, false*, etc. A GUI is a triple (W, ρ, ν) that consists of a set of widgets, a mapping $\rho : W \to 2^P$ from widgets to Test results properties, and a mapping $\nu : P \to 2^V$ from properties to values.

A GUI state is a triple (W, ρ, ω) that consists of a set of widgets, a mapping $\rho : W \to 2^P$ from widgets to properties, and a mapping $\omega : P \to V$ from properties to values.

Each GUI widget accepts as input a set of user events *E* triggered by user actions. Examples of events are *clicks, mouse moves*, etc. Formally, events accepted by a GUI widget are defined as:

$$\forall w \in W \mid \exists E_w \supseteq E : accept(w, E_w)$$

Each GUI widget has zero or more event listeners *L* registered for events performed on the widget. Event

listeners contain computational logic and are notified (their methods invoked) when the corresponding event occurs. Examples of listeners are *selection listener*, *modification listener*, etc. Formally, event listeners are defined as:

$$\forall w \in W \mid \exists L_w \supseteq L \wedge \forall l \in L_w \mid \exists E_l \supseteq E_w \wedge \forall e \in E_l \mid reg(l,e)$$

Since the user interacts with the GUI through events, a GUI test case (from the set *T* of GUI test cases) is an event sequence.

$$\forall t \in T : t = < e_1, e_2, ..., e_p >$$

We extend the definition of a GUI test case by adding the GUI states observed during the execution of the event sequence. Hence, GUI test case is composed of an event sequence and a set of GUI states.

$$\forall t \in T : t = \{< e_1, e_2, ..., e_p >, \{(W, \rho, \omega)\}\}$$

We define that a state s_1 subsumes state s_2 if the set widgets observed in state s_2 is a subset of the set widgets observed in state s_1, and the values for all properties of all the common widgets are identical.

$$\forall s_1, s_2 \in S : subsumes(s_1, s_2) \Rightarrow W_{s_1} \subseteq W_{s_2} \wedge P_1 = P_2 \wedge V_1 = V_2 \mid$$
$$P_1, P_2 \in P_{W_{s_1}} \cap P_{W_{s_2}} \wedge V_1, V_2 \in V_{W_{s_1}} \cap V_{W_{s_2}}$$

4. Barad and our approach

This section presents our approach implemented by Barad, our GUI testing framework. We present the key ideas, adopted abstractions, and processes that enable systematic test generation for GUI product lines.

4.1. GUI testing process in Barad

Since manual generation of a test suite for each member of a GUI product is ineffective and time consuming, we provide a tool support for automatic and systematic test generation for GUI product lines. Further, we exploit the common functionality shared among the individual members of these product lines. Our objective to address the four dimensional problem space in testing GUI product lines: (1) combinatorial GUI event sequences; (2) combinatorial data inputs for GUI data widgets; (3) combinatorial nature of features compositions; (4) feature interactions.

4.2. Test reusing algorithm.

The key idea of our approach is to exploit the program state to detect reusable tests. Our technique is similar to model checking [1]. We add to a GUI test case the states that have been observed during its execution (i.e. on execution of each test step) on the program for which it was generated. During the execution of a test case, generated for one member the product line, on another member, we use the observed program states and the states stored in the test case as an oracle for reusability of the test. The algorithm of detecting reusable tests is shown in Figure 3 and proceeds as follows: (1) execute a step of the replayed test – line 2; (2) take a state snapshot of the program – line 3; (3) if the state snapshot subsumes one of the states stored in the test proceed to the next test step, otherwise reject the test – line 4-6; (4) repeat step 3 for each step in the test – line 1; (5) if after executing of all steps the test is not rejects, we reused the test – line 9-10.

```
1  for (TestStep testStep:testCase.getSteps() {
2    testStep.execute();
3    State state = makeStateSnapshot();
4    if (!test.hasStateSubsumedInState(state)) {
5      test.reject();
6      return false;
7    }
8  }
9  test.reuse();
10 return true;
```

Figure 3. Test reusability detection algorithm

4.3. GUI Test generation algorithm

Our GUI testing framework, Barad, uses the test case generation algorithm in Figure 4. It is a combinatorial search that explores all combinations of widget-event pairs i.e., an event performed on a GUI widget. Test cases are an ordered list of one or more event-widget pairs – *test steps*.

Each iteration of the algorithm processes a collection of test cases known as the result set. To attempt to generate new test cases, an existing test case from the result set is processed by executing all of its test steps and then determining the set of possible next test steps.

```
1 Set uniqueTestCases = new LinkedHashSet();
2 Set candidateTestCases;
3 boolean newUniqueTestCasesFound;
4 do {
5   newUniqueTestCasesFound = false;
6   candidateTestCases = new LinkedHashSet ();
7   for (TestCase testCase : uniqueTestCases) {
8     returnToInitialState();
9     executeTestCase(testCase);
10    List nextSteps = generateNextSteps();
11    for (TestStep nextStep : nextSteps) {
12      TestCase candidateTestCase = testCase.clone();
13      candidateTestCase.add(nextStep);
14      candidateTestCases.add(candidateTestCase);
15    }
16  }
17  for (TestCase candidateTestCase:candidateTestCases) {
18    if (!pruneTestCase(candidateTestCase)) {
19      if (uniqueTestCases.add(candidateTestCase)) {
20        newUniqueTestCasesFound = true;
21      }
22    }
23  }
24 } while (newUniqueTestCasesFound);
```

Figure 4. Test generation algorithm

From the set of possible next steps, a new collection of candidate test cases is formed by adding a possible next step such that the candidates differ from the parent test case only by last test step. Each candidate test case is then executed at which point the state of the GUI is measured and compared to a set of remembered states. If the state is unique, i.e. it does not exist in the set of remembered states, the candidate test case is added to the result set; otherwise it is pruned. The algorithm continues until it determines that no new unique test cases exist.

4.4. Feature interactions detection algorithm

Figure 5 shows a pseudo code of our feature interactions detection algorithm. We can detect feature interactions during replaying a test suite generated for a member of the product line on another member of that product line. We determine the existence of a feature interaction by the completeness of successfully replayed tests. We detect, if the features composing the program for which the test suite was developed, interact with the features in the program on which the test suite is replayed. Note that the opposite is not necessary true. The algorithm proceeds as follows: (1) we execute all loaded tests – line 1; (2) if all loaded tests are successfully executed – line 2 – we detect lack of feature interaction (one way interaction) – line 3. Otherwise, a feature interaction (one way interaction) exists – line 5. Note that Barad generates a complete test suite up to a given length and we assume the replayed test suite is generated by our framework.

```
1  executeLoadedTests();
2  if (allLoadedTestExecutedSuccessfully()) {
3      setFeaturesInteraction(NON_INTERACT);
4  } else {
5      setFeaturesInteraction(INTERACT);
6  }
```

Figure 5. Feature interactions detection algorithm

4.5. Event space pruning exploration

We next present our event space exploration algorithm. Pseudo code of the algorithm is shown in Figure 6. This algorithm is a heuristic for pruning regions of the event inputs space by avoiding execution of test cases that would generate already observed GUI states. The algorithm proceeds as follows: (1) we check the type of feature relation - interacting or non-interacting – line 2; (2) if features interact, we must executed each generated test case (for reference see Section 4.2), since potentially unobserved GUI state may be reached – line 3; (3) if features do not interact, we execute only tests that contain a new widget (i.e. added in the tested GUI), since only the may generated unobserved GUI states – line 5-7.

Note that because of the way our GUI test generation algorithm works, we need to examine the entire event space again, since potentially new widgets may have been added. These new widgets would require test cases that generate all possible event sequence that include these widgets. However, since we replayed successfully and existing test suite, we know that the entire event input space that does not include the new widgets will generate only the GUI states observed during the replay phase. Hence, we do not execute such test cases.

```
1  while (!eventSpaceExplored()) {
2    if (getFeaturesInteraction() == INTERACT) {
3      currentTest.execute();
4    } else {
5      if (currentTest.hasStepForNewWidget()) {
6        currentTest.execute();
7      }
8    }
9    test.generateDescendents();
10 }
```

Figure 6. Event space pruning algorithm

5. Implementation

This section presents the main components of Barad. We also discuss some implementation details about these components.

5.1. Barad Agents

Our Barad framework consists of two collaborating agents operating on a symbolic and a concrete version of the application under test, respectively. For the purpose of this research, we utilize exclusively the *Concrete Agent*. It generates and executes tests on a concrete (i.e. standard) version of the GUI as well as provides reports for code coverage and detected errors.

5.2. Concrete agent.

The *Concrete Agent* generates tests adopting a traditional test generation approach (discussed in Section 4) and executes tests on the GUI. In contrast with conventional GUI testing frameworks, which restart the GUI after executing a test case, the agent reinitializes the application. The agent uses a client-server architectural. It is a JVM Tool Interface (JVMTI) and can detect defects via uncaught exceptions thrown by the GUI at runtime.

Figure 7 presents the architecture of the *Concrete Agent*. The main components are: (1) *Test generator*—generates test; (2) *Test executor*—executes test on the GUI; and (3) *Barad Studio*—presents visualization aids and controls the testing process;

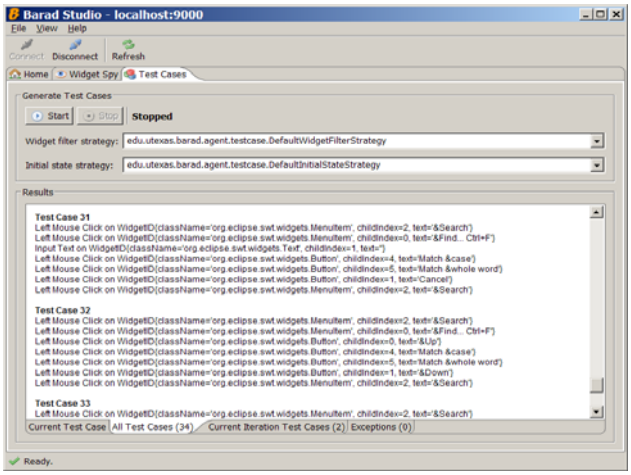

Figure 7. Screenshot of the test execution view

Figure 8 shows a screenshot of *Barad Studio* presenting a sample of the executed test cases.

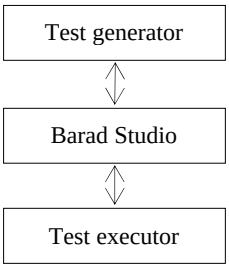

Figure 8. Concrete agent architecture

6. Related work

To the best of our knowledge previous work in GUI testing has not considered specialized approaches for testing GUI product lines. This paper presents Barad, a comprehensive framework that implements our technique to generating tests for GUI product lines. Because of the lack of related work we will discuss some approaches of testing GUIs and software product lines.

In his Ph.D. Dissertation [2] Memon presents a framework for GUI testing that generates, runs, and assesses GUI tests. This was the first framework capable of performing the whole process of test generation, execution, and result assessment for GUIs. His framework focuses on the event-flow of GUI applications. For emulating user input a specification based approach is adopted—using values from a prefilled database. The main components of the framework are presented in [2], [3], [4], [5]. The most recent research based on this framework is presented in [7] by Memon and Xie. This framework generates tests as event sequences and does not provide a mechanism for obtaining input values for data widgets.

Another approach for testing GUIs is that in which a GUI is represented as a Variable Finite State Machine from which after a transformation to an FSM, tests are obtained [6]. This is a black-box testing approach focusing on events as first class entities during the test generation phase and user inputs are not considered.

We presented in [8] Barad, our novel GUI testing framework based on symbolic execution. Barad generates values for data widgets and enables a systematic approach that uniformly addresses the data-flow as well as event-flow for white-box testing of GUI applications.

A specification-based approach for testing of software product lines is presented in [9]. This approach uses specifications written in Allow, first-order logic with sets and relations. The specifications are used for systematic test generation. The paper presents a prototype based on the AHEAD theory.

Olimpiew and Gomaa present in [10] an approach for test generation for software product lines. The key idea is mapping of a product line model to tests which are automatically generated. Metadata relevant to testing is added to the model. Authors provide as an example a Hotel product line.

7. Conclusion

We presented an approach for testing GUI product lines by reusing tests developed for individual product line members.

We define a GUI test case as an event sequence, a set of data inputs, and a set of GUI states observed during the execution of the event sequence. Such an approach allows us to detect reusable tests. We utilize the reused tests as a seed for test generation.

Our approach also detects interacting features. In case of non interacting features we achieve significant speed up of test generation. In case of interacting features we achieve the same generation speed as without test case reuse.

10. References

[1] GuillaumeBrat, Klaus Havelund, SeungJoon Park and Willem Visser '*Java PathFinder, Second Generation of a Java Model Checker*', Research Institute for Advanced Computer Science, 2000

[2] Memon, A. A comprehensive Framework For Testing Graphical User Interfaces. Ph.D. Thesis, University of Pittsburgh, Pittsburgh, 2001.

[3] Memon, A. Using Tasks to Automate Regression Testing of GUIs. In International Conference on Artificial intelligence and Applications (AIA 2004), Innsbruck, Austria, Feb. 16-18, 2004.

[4] Memon, A., Banarjee, I., and Nagarajan, A. “DART: A Framework for Regression Testing Nightly/Daily Builds of GUI Applications”. In International Conference on Software Maintenance 2003 (ICSM'03), Amsterdam, The Netherlands, Sep. 22-26, 2003, pages 410-419.

[5] Memon, A., Banarjee, I., and Nagarajan, A. What Test Oracle Should I use for Effective GUI Testing?. In IEEE International Conference on Automated Software Engineering (ASE'03), Montreal, Quebec, Canada, Oct. 6-10 2003, pages 164-173.

[6] Shehady, R., K., and Siewiorek, D., P. A Method to Automate User Interface Testing Using Variable Finite State Machines. In 27th International Symposium on Fault-Tolerant Computing, p. 80, 1997.

[7] Xie, Q., and Atif M. Memon, Using a Pilot Study to Derive a GUI Model for Automated Testing *ACM Trans. on Softw. Eng. and Method.*, 2008

[8] Ganov, S., Killmar, C., Khurshid, S., Perry, D., Test Generation for Graphical User Interfaces Based on Symbolic Execution, *Third International Workshop on Automation of Software Testing (AST)*, Leipzig, Germany 2008

[9] Uzuncaova, E., Garcia. ,D., Khurshid, S., Batory, D., A specification-based approach to testing software product lines, *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, Dubrovnik, Croatia 2007

[10] Olimpiew, E., Gomaa, H., Model-based testing for applications derived from software product lines, *Proceedings of the 1st international workshop on Advances in model-based testing*, St. Louis, Missouri 2005