Towards Systematic, Comprehensive Trace Generation for Behavioral Pattern Detection through Symbolic Execution

Markus von Detten
Software Engineering Group, Heinz Nixdorf Institute
University of Paderborn, Germany
mvdetten@upb.de

ABSTRACT

In reverse engineering, dynamic pattern detection is accomplished by collecting execution traces and comparing them to expected behavioral patterns. The traces are collected by manually executing the program under study and therefore represent only part of all relevant program behavior. This can lead to false conclusions about the detected patterns. In this paper, we propose to generate all relevant program traces by using symbolic execution. In order to reduce the created trace data, we allow to limit the trace collection to a user-selectable subset of the statically detected pattern candidates.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic Execution*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering and reengineering*

General Terms

Design, Documentation

Keywords

Pattern Detection, Execution Traces, Symbolic Execution

1. INTRODUCTION

In software engineering, reverse engineering is the analysis of software for the purpose of understanding it or recovering its documentation [2]. Pattern detection, i.e., the detection of implementations of design patterns [4] or design deficiencies (e.g., anti patterns [1]), is an important part of reverse engineering for which many approaches have been suggested. While many early approaches only used static analyses of the source code to detect patterns, there is an increasing number of techniques which combines static and dynamic analysis methods (e.g., [5, 11, 12]). They analyze the static structure of the software as well as its dynamic behavior. While the static structure is represented by the system's source code, the data documenting the behavior has to be collected first: traces of concrete program executions have to be recorded. This can be accomplished by instrumenting the program or by monitoring the program execution, e.g., via a debugging interface. The collected traces can then be compared to pre-defined behavioral patterns to draw conclusions about the behavior of pattern implementations in the software.

A trace obtained by the execution of a program is dependent on the input. It only documents one concrete execution of the software. Similar to testing, it is not feasible to cover all possible executions of the program manually and trace them in a systematic way. This leads to a severe problem for dynamic pattern detection approaches: while we want to draw general conclusions about the behavior of pattern implementations, the dynamic analysis is only based on a fraction of the possible program behavior. A common heuristic to circumvent this problem is to execute the software "long enough" and collect multiple traces by trying different executions. It is expected that traces collected in this way will contain enough relevant information to allow a statement about the patterns of interest.

This paper sketches an approach to obtain program execution traces in a systematic way by using symbolic execution [6]. We explain how patterns candidates that have been detected through static analysis with the Reclipse Tool Suite [12] can be analyzed by the symbolic execution provided by Java PathFinder [9].

The paper is organized as follows: Section 2 will briefly present the existing pattern detection capabilities of the Reclipse Tool Suite. Section 3 sketches the extension of the approach, i.e., the systematic collection of traces by symbolic execution. Section 4 discusses related work and Section 5 concludes the paper by outlining future work.

2. THE PATTERN DETECTION PROCESS

Reclipse is a reverse engineering tool suite that is being developed at the University of Paderborn [12, 13]. The pattern detection process with Reclipse is illustrated in Figure 1. A set of structural patterns and the source code of the system under analysis are the input for the static analysis. It uses graph matching in an abstract syntax graph of the software [8] to identify *pattern candidates*, i.e., probable implementations of patterns based on the static structure of the system.

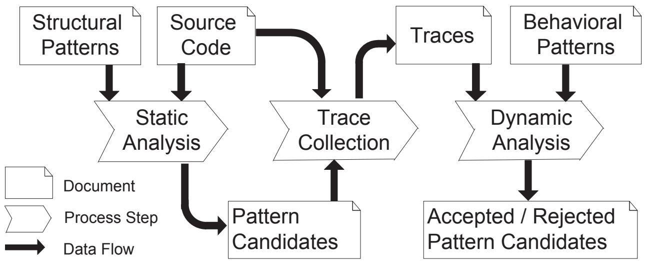

Figure 1: The current pattern detection process in Reclipse [12]

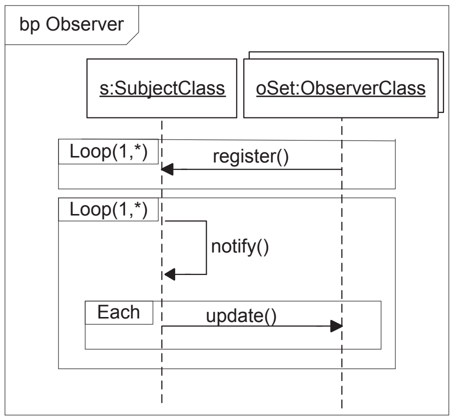

Figure 2: Observer behavioral pattern

Afterwards, traces for the candidates are collected and compared to behavioral patterns in the dynamic analysis step.

The behavioral patterns in Reclipse are specified on the basis of UML 2.0 sequence diagrams [13, 14]. Figure 2 shows the behavioral pattern for the well-known Observer pattern [4]. It describes the interaction of a subject class and a set of observer classes which are monitoring the state of the subject class. The behavioral pattern states that an observer class can register for notifications from the subject by calling the register method. Because multiple observers can register at the same subject, this can happen one to many times as indicated by the loop(1,*) fragment. Afterwards, when the subject's notify method is called, the pattern describes that the update method of each of the registered observers has to be called. This notify-update-cycle can also happen one to many times. For details refer to [13].

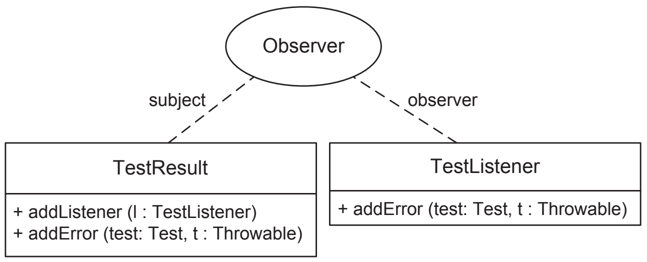
Observer
subject
observer
TestResult
+ addListener (l : TestListener)
+ addError (test: Test, t : Throwable)
TestListener
+ addError (test: Test, t : Throwable)

Figure 3: Candidate for an occurrence of the Observer pattern in JUnit 4

Figure 3 shows a candidate for the occurrence of an Observer pattern in JUnit. The static analysis detected that the class TestResult seems to play the role of the subject and the class TestListener seems to be the Observer class. The depicted methods are the relevant methods for this pattern candidate, i.e., they play the part of register (addListener), notify (TestResult::addError), and update (TestListener::addError) from the pattern.

The program is then executed in order to collect traces. However, tracing all method calls in an executed system would yield way too much data. Therefore, Reclipse only traces a filtered subset of all method calls. Only methods which appear in the behavioral patterns of the pattern candidates are traced [7] (in this case TestResult::addListener, TestResult::addError, and TestListener::addError). In the dynamic analysis step, the collected trace is compared to the behavioral patterns [15].

The dynamic analysis checks if the candidates behave according to their behavioral patterns. This analysis is based entirely on the collected traces. If the traces contain evidence that a candidate behaves only as specified by the pattern, i.e., the relevant method calls occur in an order that matches the pattern, the candidate is accepted as a true positive. If the trace shows that a candidate behaves not in accordance with the behavioral pattern, the candidate is rejected (i.e., it is assumed to be a false positive). If the trace does not contain conclusive evidence of either of these behaviors, the dynamic analysis cannot determine if the candidate is a true or a false positive.

Shortcomings of the trace collection.

Because the program execution can only cover part of all possible behavior, the collected traces do not necessarily contain behavioral data on the interesting candidates. In this case, it is very difficult to actively obtain that data because there is no straightforward way to get the system to behave as desired. The actual system behavior can normally be influenced by the input or by direct interaction with the system, e.g., via a graphical user interface. The reverse engineer on the other hand only knowns the classes and methods which play a part in the corresponding pattern candidate. Without an in-depth analysis of the system's source code, it is not possible to determine which interaction with or input into the system will trigger the execution of a particular candidate.

If the collected trace does not contain behavioral data for a certain candidate, the reverse engineer has no choice but to collect more data. He can either interact randomly with the system to trigger the desired behavior by chance, or he systematically tries to cover as much interaction as possible. Neither of these heuristics guarantees to produce the desired data. In addition, the first approach does not allow to repeat the data collection in a systematic way. The second approach requires meticulous logging of the user interaction and input data to be repeatable.

Furthermore, the current way of collecting traces does not allow definitive conclusions about the candidates. The only conclusion the reverse engineer can draw is that a given candidate behaves according to the behavioral pattern *for the part of the program that was actually executed.* The candidate might violate the pattern for another execution. The reverse engineer, however, would never know because he can only base his judgment on the collected trace data.

So, a more systematic and comprehensive way to collect the trace for the dynamic analysis could greatly improve the pattern detection process.

3. SYSTEMATIC AND COMPREHENSIVE TRACE GENERATION WITH JPF

Symbolic execution as proposed by King allows to reason about classes of program executions instead of individual, concrete executions [6]. This is accomplished by assigning symbolic values instead of concrete values to variables and evaluating the different possible control flows. Monitoring a symbolic execution of a program can therefore yield a number of traces that cover the complete possible program behavior.

```
public void m1(int x) {
    if (x == 5) m2();
    else m3(); }
```

Consider this simple program. A concrete execution of `m1` would lead to either `m2` or `m3` being called, depending on the value of `x`. Hence, the collected trace would either contain `m1` and `m2` *or* `m1` and `m3`. Executing `m1` symbolically instead (with `x` as a symbolic variable) would generate two traces: One with $x = 5$ and one with $x \neq 5$. In the first case `m1` and `m2` would appear in the trace, in the second case `m1` and `m3` would be traced.

Symbolic PathFinder, an extension for the Java Source Code model checker Java PathFinder (JPF), allows the symbolic execution of Java code [9]. We propose to use Symbolic PathFinder to generate traces for pattern candidates detected by Reclipse.

3.1 Dealing with the complexity

The most straightforward approach for the generation of execution traces is the symbolic execution of the system's main method. This way, all possible paths through the system are evaluated and all possible traces for the detected pattern candidates can be collected. This approach obviously suffers from a high computational complexity and also generates extremely large amounts of trace data. In addition, the symbolic execution of programs with graphical user interfaces is problematic. Therefore, we propose the following ideas to refine the straightforward approach.

Preselecting pattern candidates.

During the static analysis, a large number of pattern candidates can be detected by Reclipse. As explained in Section 2, only method calls that are part of the behavioral patterns are recorded in the traces. Still, generating all possible traces for every candidate will most certainly yield an impractically large amount of data. Hence, we allow the reverse engineer to select a subset of the statically detected pattern candidates. During the symbolic execution, only methods of these selected candidates are recorded in the traces which reduces the trace files to manageable sizes.

Loops.

A common problem for symbolic generation of test data is the execution of loops which obviously has to be limited for symbolic execution. This, however, is not a concern for our purpose of behavioral pattern detection. During the dynamic analysis, we only consider the execution order of methods and not the state of the program which may depend on the number of loop iterations. The behavioral pattern in Figure 2, for example, has two loop fragments. Each execution trace that contains between one and arbitrary many executions of these loops conforms to the pattern. Thus, it is sufficient to limit the symbolic execution of loops to one iteration.

Using partial symbolic execution.

Symbolic PathFinder allows to limit the symbolic execution to a number of methods and variables which can be defined by the user. The program is executed normally until one of the selected methods or variables is reached. Symbolic PathFinder then switches to symbolic execution and calculates all possible execution paths for the different variable values [10]. Because we know the pattern candidates, we can infer the methods which play a part in their execution, e.g., methods that call methods from the corresponding behavioral patterns. Thus, we can limit the symbolic execution to those methods that actually have an influence on particular pattern candidates.

User interaction.

User interaction constitutes a challenge for symbolic execution. While textual interaction via a command line can still be covered by treating the textual input as a symbolic variable, graphical user interfaces cannot be simulated easily. JPF offers the library `jpf-awt` to abstract away the rendering aspects of the GUI and still preserve the control flow of AWT- and Swing-based GUIs. The implementation is still rudimentary though, so the library does not work for all applications. However, it generally allows to apply our approach to GUI-based systems.

3.2 Interpreting the generated traces

Until now, the concrete execution of programs only allowed for very imprecise conclusions about the pattern candidates. Whether a candidate was a true or false positive could only be answered with regard to the actually executed part of the system. The use of symbolic execution enables the reverse engineer to get a much clearer picture.

Because (at least in theory) all possible behavior of the system is executed and the relevant parts are traced, a more definite statement can be made for each candidate. Even if abstraction and inaccuracy of the symbolic execution may miss some execution paths, the data is much more comprehensive than for concrete execution. The following three cases can be distinguished:

If a candidate's traces only show behavior which complies with the corresponding pattern, it is a true positive.

If at least one trace violates the behavioral pattern, a candidate has the potential to behave incorrectly. This means it is either a false positive or at least a flawed implementation of the pattern in question.

In the third case, no trace complies with the behavioral pattern but no trace violates it either. This can happen when, e.g., only one of the candidate's methods is executed at all. This may be allowed by the pattern but because no other relevant methods are executed afterwards, the pattern behavior is neither completed nor violated. For instance, the behavioral pattern in Figure 2 would not be violated if observers would register without ever being notified of a change (of course, this hints at an implementation problem). In this case, the candidate can also be regarded as a false positive because it never shows the complete correct behavior.

3.3 Validation of the approach

We validated the approach for a number of small self-made example programs with only few classes. These initial experiments showed that Symbolic PathFinder can easily be integrated with Reclipse and generates the desired traces which can then be processed by Reclipse's dynamic analysis. We were able to reliably distinguish implementations of the State, Strategy and Observer patterns.

The application of our prototype to a realistic example failed though. We tried to use our approach to generate the traces for Observer candidates in JUnit 4.8.2 (cf. Figure 3) and JRefactory 2.6.24. Much to our regret, we could not achieve results for these examples. Both programs use AWT/Swing-based user interfaces. The symbolic execution of that part of the software lead to errors in Symbolic PathFinder. As these are known errors which are currently being fixed, we are optimistic that we will soon be able to validate our approach for these examples.

4. RELATED WORK

Symbolic execution is applied for a multitude of problems in different domains. While the method was originally conceived for the generation of test input [6], it is now also applied in fields like the behavioral verification of safety-critical, embedded systems. The Java PathFinder extension Symbolic PathFinder [9] was developed for test generation and correctness checking of multi-threaded programs.

Most dynamic pattern detection approaches rely on the concrete execution of programs for the collection of traces [5, 11]. De Lucia et al. [3] apply a systematic approach for the checking of behavioral patterns. Similar to our approach, they begin with a structural analysis to identify pattern candidates. Then, they use the model checker SPIN to analyze whether the candidates can possibly show the desired behavior. This way, false negatives can be removed from the set of candidates. Afterwards, a straight-forward dynamic analysis is carried out for the remaining candidates to verify if they actually show the expected behavior at run-time. This last step has the same disadvantages as other approaches which employ concrete execution for the collection of traces.

5. CONCLUSION AND FUTURE WORK

We presented an approach to use symbolic execution for the systematic and comprehensive generation of program traces. We discussed that these generated traces will greatly improve the quality of dynamic pattern detection results. In addition, we pointed out possible problems of our method and sketched a number of heuristics to alleviate them.

Our research is still in its early stages. Our prototype implementation worked well for very small examples. A broader validation is impeded by problems in JPF at the moment which will hopefully be fixed soon. We are optimistic that the partial symbolic execution for selected candidates will reduce the complexity sufficiently to be applicable to realistic systems.

Acknowledgments

Our thanks go to Andreas Volk and Marie Christin Platenius for their ideas and their efforts in implementing the prototype.

6. REFERENCES

[1] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mombray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons, Inc., 1998.

[2] E. J. Chikofsky and J. H. Cross, II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[3] A. de Lucia, V. Deufemia, C. Gravino, and M. Risi. Improving Behavioral Design Pattern Detection through Model Checking. In *Proc. of the 14th Conf. on Software Maintenance and Reengineering*, pages 179–188. IEEE, 2010.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic Design Pattern Detection. In *Proc. of the 11th Int. Workshop on Program Comprehension*, pages 94–103. IEEE, 2003.

[6] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[7] M. Meyer and L. Wendehals. Selective Tracing for Dynamic Analyses. In *Proc. of the 1st Workshop on Program Comprehension through Dynamic Analysis*, pages 33–37, 2005.

[8] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-Based Design Recovery. In *Proc. of the 24th Int. Conference on Software Engineering*, pages 338–348. ACM, 2002.

[9] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proc. of the Int. Conf. on Automated Software Engineering*, pages 179–180. ACM, 2010.

[10] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing NASA Software. In *Proc. of the 2008 Int. Symp. on Software Testing and Analysis*, pages 15–26. ACM, 2008.

[11] K. Sartipi and L. Hu. Behavior-Driven Design Pattern Recovery. In *Proc. of the Int. Conf. on Software Engineering and Applications*, pages 179–185, 2008.

[12] M. von Detten, M. Meyer, and D. Travkin. Reverse Engineering with the Reclipse Tool Suite. In *Proc. of the 32nd Int. Conf. on Software Engineering*, volume 2, pages 299–300. ACM, 2010.

[13] M. von Detten and M. C. Platenius. Improving Dynamic Design Pattern Detection in Reclipse with Set Objects. In *Proc. of the 7th Int. Fujaba Days*, pages 15–19, 2009.

[14] L. Wendehals. Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams. In *Proc. of the 6th Workshop Software Reengineering*, volume 24 of *Softwaretechnik-Trends*, pages 63–64. GI, 2004.

[15] L. Wendehals and A. Orso. Recognizing behavioral patterns at runtime using finite automata. In *Proc. of the 4th Workshop on Dynamic Analysis*, pages 33–40. ACM, 2006.