A Static Typestate Verifier FED with Enriched Models

Immanuel Joshua.F
M.Tech , Database Engineering, Indian Institute Of Information Technology Srirangam, India.

S. Jayanthi
Assistant Professor, Department of Computer Science and Engineering,
Anna University of Technology Thiruchirappalli, India.

Abstract – Dynamic specification mining observes program executions to infer models of normal program behavior. What makes us believe that we have seen sufficiently many executions? The TAUTOKO ("Tautoko" is the Ma¯ori word for "enhance, enrich.") typestate miner generates test cases that cover previously unobserved behavior, systematically extending the execution space, and enriching the specification. To our knowledge, this is the first combination of systematic test case generation and typestate mining—a combination with clear benefits: On a sample of 800 defects seeded into six Java subjects, a static typestate verifier fed with enriched models would report significantly more true positives and significantly fewer false positives than the initial models.

Index Terms – Specification mining, test case generation, typestate analysis.

1. INTRODUCTION

In the past decade, automated validation of software systems has made spectacular progress. On the testing side, it is now possible to automatically generate test cases that effectively explore the entire program structure; on the verification side, we can now formally prove the absence of undesired properties for software as complex as operating systems [1]. To push validation further, however, we need specifications of what the software actually should do. Writing such specifications has always been hard—and, so far, prohibited the deployment of advanced development methods. A potential alternative is specification mining—i.e., extracting high-level specifications from existing code. In the context of this work, a specification is a model that is extracted from an existing program, rather than a manually written specification, which usually exists before the program is written.

To have mined specifications reflect normal rather than potential usage, dynamic specification mining observes executions to infer common properties. Typical examples of dynamic approaches include DAIKON [10] for invariants or GK-tail for object states. A typical application of mined specifications is program understanding, and under the assumption that the observed runs represent valid usage, the mined specifications also represent valid behavior. The common issue of specification mining techniques, though, is that they are limited to the (possibly small) set of observed.

To address this problem, we use test case generation to systematically enrich dynamically mined specifications. Combined this way, both techniques benefit from each other can profit from mined specifications, as their complement points to behavior that should be explored. The goal of this work is to explore the extent to which the quality of dynamically mined specifications can benefit from generated tests.

In a nutshell, we leverage our earlier work [8], [9] to dynamically mine typestate specifications—finite state automata describing transitions between object states. The initially mined specification contains only observed transitions (Section 2). To enrich the specification, our TAUTOKO tool generates test cases to cover all possible transitions between all observed states, and thus extracts additional states and transitions from their executions. These transitions can either end in legal states, thus indicating additional legal interaction, or they can raise an exception, thus indicating illegal interaction. Discovering such illegal interactions is the biggest advantage of our approach, as exceptional behavior is rarely covered by conventional executions or tests. How can we assess the benefits of such enriched specifications? For this purpose, we put them to use in static typestate verification. Typestate verification statically discovers illegal transitions. Its success depends on the completeness of the given specification: The more transitions are known as illegal, the more defects can be reported; and the more transitions are known as legal, the more likely it is that additional transitions can be treated as illegal. We expect that our enriched specifications are much closer to completeness than the initially mined specification On a sample of 800 defects seeded into six Java subjects, we show that our static typestate verifier fed with enriched models reports significantly more true positives than when being fed with the initial models.[1] We expect this increased accuracy to generalize toward arbitrary uses of mined specifications, and thus conclude (Section 6) that test case generation is a useful method to enrich dynamically mined specifications.

This paper extends an earlier version presented at ISSTA 2010

[7]. The previous version used a test case generation that was limited to mutation of existing tests. This paper includes an alternative approach which does not require existing test cases. Starting with an automatically generated initial test suite satisfying a standard criterion such as branch coverage, we iteratively derive new test cases from the typestate automaton to systematically explore the behavior of the considered target class. In this paper, we compare the performance of both techniques and discuss advantages and disadvantages of each approach.

2. MINING TYPESTATES

A typestate automaton (or simply typestate) is a finite state automaton which encodes the legal usage of a class under test (CUT). Its states represent different states of an object, and transitions are labeled with method names. As an example, consider the typestate for the SMTPProtocol class from the ristretto [7] library. After initialization, an SMTPProtocol object is in its initial state 0; calling open-Port() brings it into state 1; and calling quit() from this state brings it back into the initial state 0.

If an invocation of method m in state s causes an exception, the typestate contains a transition from s to a special state ex labeled with m. In our example, this is the case if quit() is invoked from the initial state 0; this raises a NullPointerException. A static typestate verifier can take this very specification and check a client for con-formance; if it is possible to invoke quit() while still being in the initial state 0, the verifier will flag an error.

To obtain such typestate specifications from programs, we leverage the ADABU tool presented in earlier work [8], [9]. ADABU mines so-called object behavior models that capture the behavior of objects at runtime. A behavior model for an object o is a nondeterministic finite state automaton where states are labeled with the values of fields that belong to o, and transitions occur when a method invoked on o changes the state. Fig. 3 shows an object behavior model for an instance of SMTPProtocol. This model was mined by ADABU from an execution of the regression test suite for SMTPProtocol.

To mine such models from a program run, ADABU instruments the program such that the values of all fields from an instance of the CUT are captured after every method call. Each set of field values uniquely describes the state of the object, and the sequence of states and method calls uniquely describes the object behavior model for the observed instance of the CUT. The advantage of using field values to label states is that equivalent object states are easy to detect, since they have the same labeling. Other automata-based specification mining techniques [6] have to resort to heuristics to solve this problem, which may lead to overgeneralizations in the learned automata.

The challenge with using concrete field values is that the number of states in a model may become very large, and the models are difficult to compare. In the example in Fig. 3, the value of variable socket is the object identifier for the Socket object. In another run of the program, this value might be different due to scheduling, and hence the states would be different in both models.

However, this difference is not important both for the behavior of the class. Instead, what matters is that the socket is not null so that the protocol is able to communicate with the server. Hence, instead of using concrete values, ADABU uses abstract values: Complex fields are mapped to null or not null, numerical fields are mapped to less than, equal to, or larger than zero, and Boolean fields remain unchanged.

This approach is based on work of Liblit et al. [9] that uses the same abstractions in the context of bug localization. State abstraction reduces the size of the models and makes them comparable across program runs. However, this technique also entails a loss of information, as the abstract models are less detailed than the concrete models. In our experience, the above approach provides a good tradeoff between the size of a model and its expressiveness. In this paper, we use ADABU to mine object behavior models and then convert them into typestate automata. The idea for this approach is based on the observation that typestates and object behavior models are closely related. The two main differences are as follows:

State. In typestate automata, states are anonymous; in object behavior models, they are labeled with the values of fields.

Exceptions. Typestates represent failing method calls by transitions to a special state ex. In object behavior models, information about exceptions is only stored at edges.

1. The automaton is initialized with two states labeled start and ex.
2. Each state s of the behavior model is assigned a unique number n, and a corresponding state labeled n is added to the typestate.
3. For each invocation of a method m between two states s_i and s_j, a new transition labeled with m is added to the typestate: If the invocation raised an exception, the transition is added from s_i to ex, otherwise it is added from s_i to s_j.

The typestate introduced earlier was not specified manually, but automatically obtained from the object behavior model. For the remainder of this paper, we will use the term "mine typestate automata" to summarize the process of mining object behavior models and converting them to typestate automata.

3. ENRICHING TYPESTATES

To yield precise results and few false positives during verification, a typestate needs to be complete, i.e., it needs to contain all relevant states and transitions for all methods in all states. To test TAUTOKO, we ran it on a set of projects and

mined typestates from the test suite executions for a set of interesting classes. Unfortunately, for the investigated classes we found that most typestates only contained a fraction of all transitions. In particular, most typestates were missing transitions for failing methods, which renders mined typestates useless for typestate verification.

We believe that the lack of observed failures is an issue that is common to many projects—and thus affects every approach for dynamic specification mining:

1. Most defects due to wrong usage of a class raise exceptions and are therefore easy to detect and fix. Thus, a specification miner tool will seldom record misuse and exceptions when tracing normal application executions.
2. Unfortunately, we observed the same problem of missing exceptions when tracing test suites. Most developers do not test for exceptions. One explanation for this is that triggering an exception often only covers a few lines.
3. To generate a complete model, lots of tests are required. Usually, developers do not have enough time to write so many tests. Also, developers tend to skip tests which they consider to be too obvious or are convinced that they should work.

One way to approach this problem is to use test case generation to create new tests that execute previously unknown states and transitions. The general idea of combining specification mining with test case generation was first described by Xie and Notkin [3].

In this paper, we extend the original idea to generate tests specifically targeted at enriching typestate automata. There is a huge variety of test generation strategies, ranging from complex static analyses such as symbolic execution [8] to simple random testing techniques [5].

3.1. Mutating Existing Test Cases

Our initial technique works as follows: In the first step, TAUTOKO executes the test suite and mines a model for the CUT. This model is called the initial model. After that, it attempts to generate mutations to the test suite such that all methods are executed in all states of the initial model.

TAUTOKO then applies each mutant in isolation and mines new models from the execution of the modified test suite. Finally, the initial model and all new models are combined into the model for the CUT. The advantage of this approach is that it allows to use the vanilla ADABU tool to mine models from each test, and TAUTOKO only has to take care of generating tests and combining models. For every method m that expects parameters other than the receiver (lines 6-13), TAUTOKO finds all invocations of m in the initial typestate (line 7), tries to find a path that leads to s, and creates a mutated test that suppresses all method calls along the path (line 11).

To demonstrate the effect of TAUTOKO, which shows the initial model of class SMTPProtocol mined from an execution of the project's test suite. In contrast, the enriched model generated by TAUTOKO after evaluating all mutations. Not only does the enriched model contain several additional transitions, but it now also explicitly lists the exceptional behavior in its ex state. We will use these models to illustrate the techniques presented in this section.

Mutant generation starts by statically determining the set of methods that belong to the CUT or one of its supertypes. For every such method m, TAUTOKO tries to generate mutations such that m is invoked in all states of the initial model. To invoke method m in states, TAUTOKO will either add an invocation of m, or suppress one or more existing method invocations. The choice of adding or deleting invocations depends on the number and types of the parameters m expects.

If m only requires a reference to the receiver object, TAUTOKO simply adds a new call to m right after a method call that caused a transition to s in the initial model. For example, in Fig. 4, to invoke method dropConnection() in state 1, TAUTOKO adds a call to dropConnection() right after the call to openPort() that causes the transition to state 1.

A problem arises if m expects parameters beyond the receiver object. In this case, we need to provide values for the parameters in order to call m. Our initial approach is to reuse existing invocations of m. If the initial model contains an invocation of m in another state t, TAUTOKO suppresses method calls such that the call occurs in state s instead. For example, to call method authSend(byte[]) in state 0, we can suppress the invocation of openPort() that causes the transition from state 0 to 1. If there is more than one possibility, TAUTOKO generates tests for each possibility.

The advantage of this approach is that it is simple to implement and works also for complex parameters that are difficult to generate. However, this approach also has several limitations:

1. Our technique is unable to handle methods with parameters that are never invoked by the program. Since we do not synthesize parameter values, TAUTOKO is unable to generate calls to these methods.
2. In order to invoke a method m which takes parameters in states, the test case needs to contain a call to m on a path before states is reached. Otherwise, TAUTOKO is unable to modify the test accordingly.
3. If TAUTOKO suppresses a method call with a return value, it has to substitute the result of the call with default values such as null or false. In some cases, this breaks the test case and ADABU cannot observe the method call in the desired state.

Overall, to be able to reliably call methods which take parameters, we need to apply more generic test generation

schemes. However, despite the above limitations, our evaluation results show that even with the simple mutation-based approach, enriched specifications already contain much more information and are likely to be much more useful in any verification setting, iterates over all states s of the initial typestate. For every method m that expects parameters other than the receiver (lines 6-13), TAUTOKO finds all invocations of m in the initial typestate (line 7), tries to find a path that leads to s, and creates a mutated test that suppresses all method calls along the path (line 11). If the sole parameter to m is the receiver (lines 14-19), TAUTOKO finds all transitions after which the object is in state s (line 15) and generates a new test that invokes m right after the call that caused the transition (line 17). The final loop (lines 23-26) executes all tests, mines new typestates from each execution, and merges the new typestate into the current version. After the loop has finished, the procedure returns the enriched typestate.

3.2. Test Case Generation Using Typestate Automata

The improvements achievable by mutating test cases depend to a large extent on the type and quality of the already existing test cases—a simple test case mutation approach can only add new method calls for which all parameter dependencies are satisfied. To overcome this limitation, a full-fledged test generation approach can be employed, such that new objects are generated as necessary.

Using automatic test generation to bootstrap the process is not only convenient when no previously written test cases are available, automatically generated test cases can have very high coverage and include a significant amount of exceptional behavior. Thus, even when there is an existing test suite, additional automatically generated test cases might provide useful information.

4. CONCLUSIONS

Dynamic specification mining is a promising technique, but its effectiveness entirely depends on the observed executions. If not enough tests are available, the resulting specification may be too incomplete to be useful. By systematically generating test cases, our TAUTOKO proto-type explores previously unobserved aspects of the execution space. The resulting specifications cover more general behavior and much more exceptional behavior.

An evaluation with six different subjects shows that TAUTOKO is able to enrich specifications with new transitions in all cases. With enriched specifications, a typestate verifier produces significantly more true positives, but the false positive rate increases due to the nondeterminism of the underlying model miner. We showed that systematic test case generation can iteratively learn new states and transitions, further improving the number of detected defects. A potential higher number of false negatives illustrates one of the current problems of auto-mated test case generation—generated tests do not resemble real usage. Generally, we expect test case generation to be applicable to all techniques of dynamic specification mining, improving the effectiveness of mined specifications.

We can also generate new executions (test case generation) or even change their code (mutation analysis). The interplay of these techniques brings lots of opportunities for exciting research topics illustrates one of the current problems of auto-mated test case generation—generated tests do not resemble real usage. Generally, we expect test case generation to be applicable to all techniques of dynamic specification mining, improving the effectiveness of mined specifications. Verification still produces a considerable number of false positives, which are partly due to technical limitations of our typestate verifier implementation. Once a sound analysis such as WALA becomes publicly available, we expect better performance that would make typestate verification using mined models applicable in practice.

Despite the improvements by TAUTOKO, typestate verification still produces a considerable number of false positives, which are partly due to technical limitations of our typestate verifier implementation. Once a sound analysis such as WALA becomes publicly available, we expect better performance that would make typestate verification using.

REFERENCES

[1] J.C. King, “Symbolic Execution and Program Testing,” Comm. ACM, vol. 7, no. 3, 215-249, 1976.
[2] B. Liblit, A. Aiken, A.X. Zheng, and M.I. Jordan, “Bug Isolation via Remote Program Sampling,” ACM SIGPLAN Notices, vol. 38, no. 5, pp. 141-154, May 2003.
[3] D. Lorenzoli, L. Mariani, and M. Pezze`, “Automatic Generation of Software Behavioral Models,” Proc. 30th Int’l Conf. Software Eng., pp. 501-510, 2008.
[4] R. Majumdar and K. Sen, “Hybrid Concolic Testing,” Proc. 29th Int’l Conf. Software Eng., pp. 416-426, 2007.
[5] P. Mcminn, “Search-Based Software Test Data Generation: A Survey,” Software Testing, Verification, and Reliability, vol. 14, pp. 105-156, 2004.
[6] A. Mesbah and A. van Deursen, “Invariant-Based Automatic Testing of AJAX User Interfaces,” Proc. IEEE 31st Int’l Conf. Software Eng., pp. 210-220, 2009.
[7] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid, “Korat: A Tool for Generating Structurally Complex Test Inputs,” Proc. 29th Int’l Conf. Software Eng., pp. 771-774, 2007.
[8] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, “Static Specification Mining Using Automata-Based Abstractions,” Proc. Int’l Symp. Software Testing and Analysis, pp. 174-184, 2007.
[9] R.E. Strom and S. Yemini, “Typestate: A Programming Language Concept for Enhancing Software Reliability,” IEEE Trans. Software Eng., vol. 12, no. 1, 157-171, 1986.
[10] P. Tonella, “Evolutionary Testing of Classes,” SIGSOFT Software Eng. Notes, vol. 29, no. 4, 119-128, 2004.
[11] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann, “Online Testing with Model Programs,” SIGSOFT Software Eng. Notes, vol. 30, no. 5, 273-282, 2005.
[12] A. Wasy lkowski, A. Zeller, and C. Lindig, “Detecting Object Usage Anomalies,” Proc. Sixth Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. the Foundations of Software Eng., pp. 35-44, 2007