Generating Test Cases for Specification Mining

Valentin Dallmeier · Nikolai Knopp · Christoph Mallon · Sebastian Hack · Andreas Zeller
Saarland University – Computer Science
Saarbrücken, Germany
{dallmeier, knopp, mallon, hack, zeller}@cs.uni-saarland.de

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[1] 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.

Categories and Subject Descriptors

D.2.1 [**Software**]: Software Engineering—*Software/ Program Verification, Requirements/Specifications*

General Terms

Algorithms, Documentation, Reliability, Verification

Keywords

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. 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. Mined specifications can be used for program understanding, but also for formal verification or regression testing.

To have 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 [18] for object states. The common issue of these approaches, though, is that they are limited to the (possibly small) set of *observed* executions. If a piece of code is not executed, it will not be considered in the specification; if it is executed only once, we do not know about alternative behavior.

To address this problem, we use *test case generation* to systematically *enrich dynamically mined specifications.* Combined this way, both techniques benefit from each other: Dynamic specification mining profits from test case generation, since additional executions can be observed to enrich the mined specifications. Test case generation, on the other hand, can profit from mined specifications, as their complement points to yet unobserved behavior.

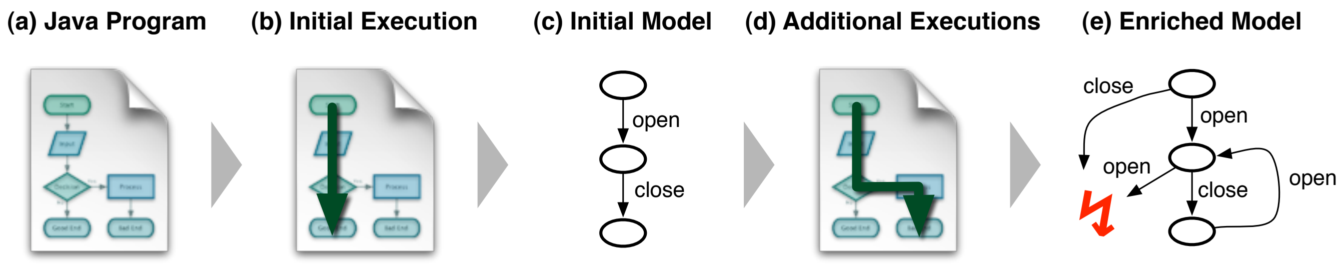
In a nutshell, our approach works as follows (see Figure 1). We leverage our earlier work[7, 8] to dynamically mine *typestate specifications*—finite state automata describing transitions between object states. The initially mined specification contains only observed transitions (Section 2). To enrich the specification, our TAUTOKO tool generates test cases to cover all possible transitions between all observed states, and thus extracts *additional transitions* from their executions (Section 3). 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 specifications; and therefore, the static verifier should be much more accurate in its reports.

This hypothesis is confirmed by an experiment (Section 4): On a sample of 800 defects seeded into six Java subjects,

[1] "Tautoko" is the Mãori word for "enhance, enrich".

ISSTA'10, July 12–16, 2010, Trento, Italy.



(a) Java Program (b) Initial Execution (c) Initial Model (d) Additional Executions (e) Enriched Model

Figure 1: TAUTOKO overview. TAUTOKO takes an executable JAVA program (a) and observes its execution (b) to extract an initial typestate model (c). It then generates additional executions (test cases) to cover missing model transitions (d). The additional observed behavior results in an enriched specification (e).

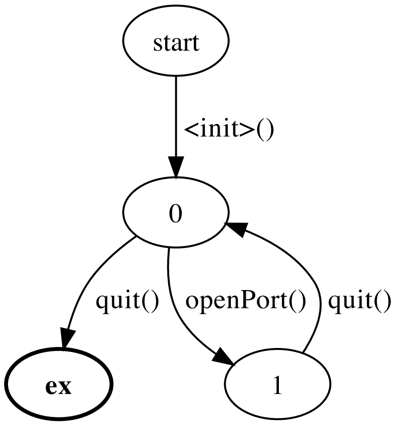


Figure 2: Typestate for `SMTPProtocol`. The failing call to `quit()` shows as a transition to *ex*.

we show that our static typestate verifier fed with enriched models reports significantly more true positives, and significantly fewer false positives than when being fed with the initial models.[2] We expect this increased accuracy to generalize towards arbitrary uses of mined specifications, and thus conclude (Section 6) that test case generation is a useful method to enrich dynamically mined specifications.

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 Figure 2, showing the typestate for the `SMTPProtocol` class from the ristretto [16] library. After initialization, an `SMTPProtocol` object is in its initial state 0; calling `openPort()` 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 conformance; 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 [7, 8]. ADABU mines so-called *object behavior models* that capture the behavior of objects at runtime. A behavior model for an object o is a 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. Figure 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`.

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.

Since both types of models are based on finite state automata, it is easy to convert an object behavior model to a typestate. We therefore use ADABU to mine behavior models, and convert them to typestates afterwards. Converting behavior models into typestate automata is straightforward and essentially consists of the following three steps:

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 in Figure 2 introduced earlier was not specified manually, but automatically obtained from the object behavior model in Figure 3.

[2] In the remainder of the paper, we will use the terms “specification” and “model” interchangeably.

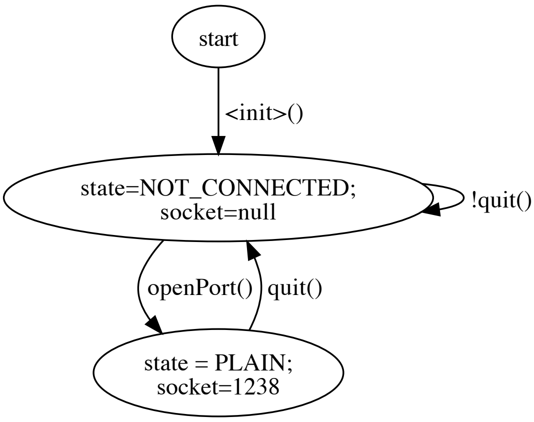


Figure 3: Object behavior model mined by ADABU from an execution of the regression test suite. Calls that raise an exception (such as `quit()`) are marked with "!". This model can be automatically converted into the typestate in Figure 2.

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:

- Most defects due to wrong usage of a class raise exceptions and are therefore easy to detect and fix. Thus, a specification miner will seldom record misuse and exceptions when tracing normal application executions.
- 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.
- 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 [32]. 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 [29] to simple random testing techniques [5, 22]. In this work, we use a test generation strategy that generates new tests by *mutating an existing test suite.*

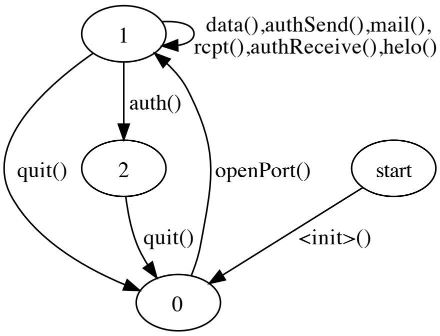


Figure 4: An initial model of the `SMTPProtocol` class as mined from the unmodified regression test suite.

Our 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.

To demonstrate the effect of TAUTOKO, consider Figure 4 which shows the initial model of class `SMTPProtocol` mined from an execution of the project's test suite. In contrast, Figure 5 shows 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 super types. 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 state s, 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 Figure 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 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.

The advantage of this approach is that it is simple to

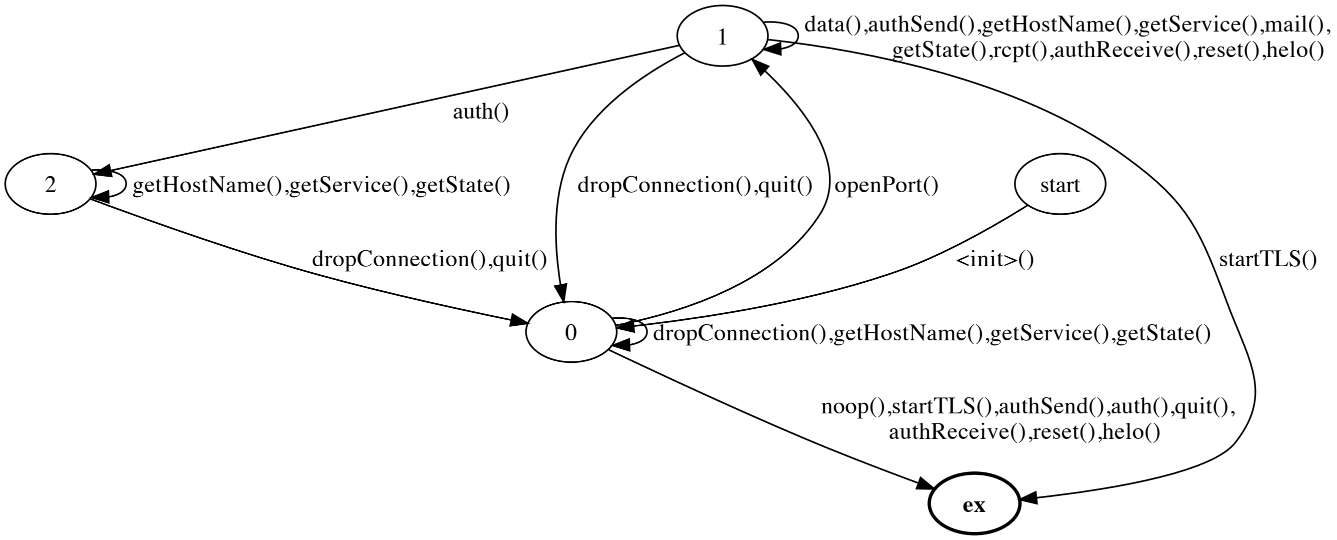

1
data(),authSend(),getHostName(),getService(),mail(),
getState(),rcpt(),authReceive(),reset(),helo()
auth()
2
getHostName(),getService(),getState()
dropConnection(),quit()
openPort()
start
dropConnection(),quit()
<init>()
startTLS()
0
dropConnection(),getHostName(),getService(),getState()
noop(),startTLS(),authSend(),auth(),quit(),
authReceive(),reset(),helo()
ex

Figure 5: Enriched model of the `SMTPProtocol` class. Compare with the initial model in Figure 4.

implement and works also for complex parameters that are difficult to generate. However, our approach is unable to handle methods with parameters that are never invoked by the program. To call such methods, we would need more generic test generation schemes [22, 28]. Still, our evaluation results show that even with this simple approach, enriched specifications already contain much more information and are likely to be much more useful in any verification setting.

Algorithm 1 shows pseudo code for the procedure to enrich a typestate for class c. Input to the algorithm consists of the test suite, the initial typestate and the set of methods that can be called on c. The main loop of the algorithm (lines 3-23) iterates over all states s of the initial typestate. For every method m that expects parameters other than the receiver (lines 7-14), TAUTOKO finds all invocations of m in the initial typestate (line 8), tries to find a path that leads to s, and creates a mutated test that suppresses all method calls along the path (line 12). If the sole parameter to m is the receiver (lines 16-20), TAUTOKO finds all transitions after which the object is in state s (line 16) and generates a new test that invokes m right after the call that caused the transition (line 18). The final loop (lines 25-28) 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.

4. EXPERIMENTAL EVALUATION

In this section, we investigate how well our approach works in practice. Our goal is to compare the usefulness of *enriched models* versus *initial models* as well as manually generated *complete models*, and thus investigate the benefits and potential drawbacks of our approach.

4.1 Subjects

To evaluate the effectiveness of TAUTOKO, we have applied it to six different JAVA subjects listed in Table 1. Altogether, we chose 6 different classes for which we generated and evaluated typestate automata. Three classes (upper half in Table 1) are part of publicly available libraries, whereas the remaining classes are part of the JAVA standard API. In terms of domain, the subjects can be divided into *I/O* (`javamail`, `ristretto`, `socket` and `zip`) and *security* (`javax.security` and `signature`).

We chose our subjects by investigating a subset of open-source projects from big hosting sites such as `Sourceforge` and `java.net`, as well as classes from the JAVA standard API. We included subjects that met the following criteria:

1. The API documentation of the class explicitly or implicitly mentions *restrictions on the order of method invocation.* In other words, we made sure that our subjects are complex enough to yield interesting specifications.

2. As a source for test runs, we solely rely on *executions as provided* by the developers of the subject class. This is to avoid introducing additional bias with self-constructed test-cases. For the first three subjects in Table 1, we use sample executions and regression test suites provided by the respective projects. We made sure that these runs cover all essential methods of the subject class. For the JAVA standard classes in our evaluation, we use conformance tests of the APACHE HARMONY project. This project aims at providing an open-source alternative to the JAVA standard classes, and therefore has a sophisticated test suite to ensure compliance with the original implementation by SUN.

3. To conduct the evaluation using the static typestate verifier, we needed an additional application for each subject that uses the subject class in its implementation. To find such applications, we searched the web using `koders.com` and `google code` search engines. To qualify for our evaluation, a project had to offer a minimum level of maturity and provide a test run that executes the subject class (See Section 4.4.2 for a rationale).

We are aware that our selection process creates a bias towards complex classes and well-tested projects. However,

Algorithm 1 Enrich Typestate Automaton

Require: Test Suite $T = (t_1, \ldots, t_n)$
Require: Initial typestate $M_{\text{init}} = (V_{\text{init}}, E_{\text{init}})$
Require: Methods to investigate M
Ensure: Enriched Typestate $M_{\text{final}} = (V_{\text{final}}, E_{\text{final}})$

```
procedure ENRICH(T,$M_{\text{init}}$)
    $T' = \{\}$
    for all $s \in V_{\text{init}} \setminus \{\text{start}\}$ do
        $M_s \leftarrow \{\text{Methods invoked in } s\}$
        for all $m \in \{M \setminus M_s\}$ do
            if hasParameters($m$) then
                $S_m \leftarrow \{s \in V_{\text{init}} \mid \exists(s, s', n) \in E_{\text{init}} : m = n\}$
                for all $u \in S_m$ do
                    $p \leftarrow \text{getPath}(u, s)$
                    if length($p$) $< \infty$ then
                        $T'$.add(suppressAllCalls($T, p$))
                    end if
                end for
            else
                $T_s \leftarrow \{(s, s', n) \in E_{\text{init}}\}$
                for all $t \in T_s$ do
                    $T'$.add(addCall($T, t, m$))
                end for
            end if
        end for
    end for
    $M_{\text{final}} \leftarrow M_{\text{init}}$
    for all $t \in T'$ do
        $M_{\text{new}} \leftarrow \text{run}(t)$
        $M_{\text{final}} \leftarrow \text{merge}(M_{\text{new}}, M_{\text{final}})$
    end for
end procedure
```

the purpose of this evaluation is not to evaluate the usage of mined specifications in general. Instead, we study how our approach for enriching mined specifications improves quality and applicability of the specifications. Section 4.5 provides a detailed discussion of threats to the validity of our results.

4.2 Enriching Models: Quantitative Evaluation

In this section, we provide a *quantitative evaluation* of our technique for enriching mined specifications. For every subject, we mine an initial model (see Section 2) from the execution of the test suite. Afterward, we use TAUTOKO to mutate the test suite and mine an enriched model. To quantify the difference between the two versions, we count the number of states and the number of transitions. A transition in this context means a method call. Since we are mostly interested in *exceptional* behavior, we also measure the number of exceptional transitions. The results of the quantitative evaluation are summarized in Table 2.

For `SMTPProtocol`, we also provided the initial model in Figure 4 and the enriched version in Figure 5[3]. Both versions have the same number of states. However, the enriched version has about three times as many transitions. Also, the initial model has no exceptional transitions, compared to 9 transitions in the enriched version.

Table 1: Subjects used in the case studies.

Subject	Type	Description
javamail	`SMTPTransport`	Sending mails via smtp.
javax. security	`LoginModule`	User authentication.
ristretto	`SMTPProtocol`	Sending mails via smtp.
signature	`Signature`	Handling of digital signatures.
socket	`Socket`	Network communication.
zip	`ZipOutputStream`	File compression with zip algorithm.

Applied to all subjects, TAUTOKO discovers new states for three out of six subjects, and significantly increases the number of transitions for all of them. None of the initial models for the first three subjects has exceptional transitions, hinting at a low quality of the test suite. This is a general trend we observed in many projects, as discussed earlier. For each of those subjects, TAUTOKO discovers new transitions that trigger exceptions. Initial models for the JAVA API classes already contain transitions to the error state. Obviously, the conformance tests of the HARMONY project also test for expected negative behavior. For the API subjects, TAUTOKO significantly increases the number of both exceptional and normal transitions. The largest relative increase is observed for `socket`, with a total of 55 exceptional transitions compared to only 2 in the initial model.

Overall, applying TAUTOKO leads to larger models with significantly more transitions. In the next section, we investigate if TAUTOKO also improves the quality of the mined specifications.

4.3 Enriching Models: Qualitative Evaluation

In this section, we take a look at how well the initial and enriched models reflect the *complete model* of the class. To this end, we compare the mined models with complete usage models. Since there are no models available for our subjects, we had to manually create them. To create the models, we investigated the source code to build a mental model that was translated into a typestate. In a few cases it was difficult to reliably judge if a method could be called in a certain state. To clarify those cases, we wrote small test cases that resolved the issue.

One problem with manual model generation is how to deal with unchecked exceptions. In JAVA, many instructions may cause null pointer dereferences or illegal array accesses. Including transitions for all those exceptions would introduce a high degree of non-determinism, which essentially renders the model useless. We therefore only include transitions for checked exceptions.

Manually creating models involves a lot of human effort (which, of course, is why we wanted to build TAUTOKO in the first place.) We therefore restricted our evaluation to only three subjects, namely `SMTPProtocol`, `ZipOutputStream` and `Signature`. In total, we spent over 10 hours on creating the specifications, where most of the time was spent on `SMTPProtocol`, which is also the most complex. Due to space restrictions, we cannot depict the complete specifica-

[3] Models for the remaining subjects are available online at the website given in Section 6.

Table 2: Enriched models have more transitions, and many more exceptional transitions.

		Original model			Enriched model		
Subject	Mutations	States	Transitions	Exceptional Transitions	States	Transitions	Exceptional Transitions
javamail	61	6	5	0	13	48	2
javax.security	9	6	5	0	6	14	6
ristretto	55	5	11	0	5	33	9
signature	23	5	30	8	5	39	13
socket	540	11	35	2	17	251	55
zip	145	11	24	5	14	62	18

tions here. However, they are available for download at the address given in Section 6. Table 3 lists structural details of the manually mined models.

Table 3: Manually specified typestate models.

Subject	States	Transitions	Exceptional Transitions
ristretto	7	86	29
signature	5	48	12
zip	6	31	9

To investigate whether TAUTOKO also improves the quality of the mined specifications, we compared initial and enriched models against the complete model:

ristretto The complete model has two more states than the initial and the enriched models. The two additional states are related to sending mails, which requires a call to initiate the mail, followed by several calls to set receivers, and a final call to send the mail. No state-based specification miner can detect this protocol, since relevant state information is transmitted to the server and is not kept locally. Apart from this, all states and transitions of the initial model are also reflected in the complete model.

The enriched model adds twenty valid transitions and nine exceptional transitions. Two exceptional transitions are invalid according to the complete model. They are caused by limitations of the mock server, which is used in the test suite of `SMTPProtocol`. This shows a limitation of our completion technique: TAUTOKO may break the boundaries of the test suite and generate invalid transitions.

signature The initial and the enriched models have the same number of states. All transitions in the initial model are in accordance with the complete model. TAUTOKO adds nine additional transitions, five out of which are exceptional transitions. All transitions are also reflected in the complete model. In total, the enriched model misses six transitions. This is due to the way TAUTOKO injects and suppresses method calls, which prevents some methods from being called in certain states.

zip The complete model has many fewer states than both the initial and the enriched model. This occurs because states in the model miner also include values for fields that are irrelevant for the usage of the object, such as `comment` or `method`. The initial model essentially contains the structure of the complete model twice, once with `method` set and once without. The enriched model contains additional states with `comments`. Despite the blow-up, the mined models are still useful since they capture all exceptional transitions of the complete model.

In summary, we found that the specification miner in combination with TAUTOKO generates valid specifications compared to manually deduced models. Like any test case generation technique, TAUTOKO cannot guarantee to cover all possible transitions; and this limitation also holds for the present subjects. Section 6 presents ideas for future work to improve coverage. In one case, TAUTOKO generates transitions that do not match the complete model. This is due to restrictions which are inherent to the general technique of enriching models by manipulating an existing test suite.

4.4 Are enriched models more useful in practice?

Results of the previous sections show that applying TAUTOKO yields better specifications. However, we would also like to know if this improvement matters in practice. To investigate this, we ran a *static typestate verifier* on a set of randomly generated defects and compared the results for initial and enriched models. For `ristretto`, `signature` and `zip`, we also included complete models from the previous section. The evaluation setting is summarized in Figure 6 and detailed in the following sections.

4.4.1 Experimental setting

Our experiment assumes the following situation: A developer starts building an application and uses classes from a library l that are unknown to her. To help the developer avoid bugs due to incorrect usage of those classes, her IDE supports lightweight typestate verification. Whenever the developer changes a method that uses classes of l for which a specification is available, the IDE launches the typestate verifier. The verifier then analyzes all changed methods and looks for incorrect usage of classes; if it finds a violation, it is presented to the user. Obviously, we would like to catch as many defects (true positives) and report as few false alarms (false positives) as possible.

To simulate the above situation in a controlled experiment, we take the following steps:

1. For each subject used in the evaluation so far, we find an *application* that uses the subject. We also require

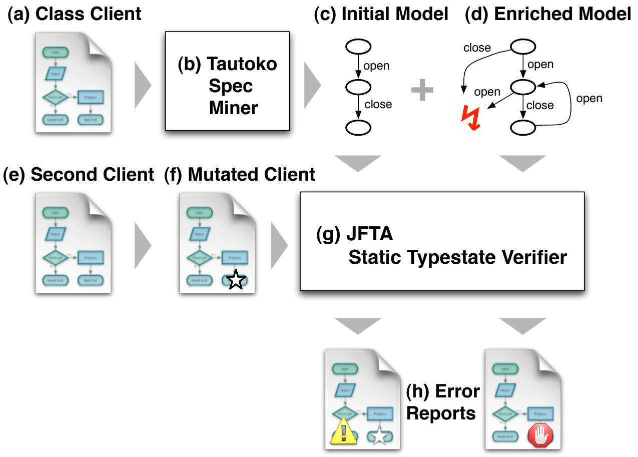


Figure 6: Evaluation overview. We take the client (a) of a class and use TAUTOKO (b) to mine both the initial model (c) and the enriched model (d). We then take a second client (e) of the same class and seed in a defect (f). The JFTA (see Section 4.4.2) static typestate verifier (g) then produces error reports (h) for the mutated client using both the initial model and the enriched model. Were available, we also include complete models (see Section 4.3). We compare the error reports in terms of true positives and false positives.

the application to provide a test suite or other means to execute the program.

2. We use our mutation tool to simulate changes a developer might make to the application. To this end, we generate *mutants* that randomly inject or suppress method calls to instances of the subject class in the application.

3. For each mutated version, we execute the *test suite* of the application to classify mutants. Mutants that raise an exception at runtime are defects that we would like a typestate verifier to detect. Mutants that do not raise an exception use the class correctly, and therefore the verifier should not report a warning.

4. Finally, we run the verifier for each mutated version to analyze all methods touched by the mutant and remember all reported violations. We use the generated mutants to measure how often the verifier points to a method invocation that actually triggers an exception (true positive), and how often the verifier reports a violation although the program runs without producing an error (false positive).

The purpose of this experiment is to measure the effect of using enriched specifications as generated by TAUTOKO over using initial specifications produced by the test suite. We therefore repeat step 4 with initial models generated by the test suite and enriched models generated by TAUTOKO. For three subjects, we also include results for the complete models created for the qualitative evaluation of Section 4.3.

We ran our evaluation for the same set of subjects used for the previous experiments. Table 4 lists the test sources for generating models, as well as the names of all applications used in the evaluation.

4.4.2 The JFTA Static Typestate Verifier

Unfortunately, existing typestate verifiers are either unavailable to the public [11], or require additional input [4]. We have therefore implemented our own typestate verifier called JFTA. JFTA is a partially inter-procedural, flow- and context-sensitive typestate verifier for JAVA classes. Input to JFTA consists of the program's byte code, a set of typestate automata, and a set of methods which are to be analyzed. In contrast to other tools such as Plural [4], JFTA does not require the programmer to provide annotations of the program code.

The core part of JFTA consists of a conservative dataflow analysis algorithm. Aliasing information is calculated using a demand-driven points-to analysis [26]. As the primary focus of JFTA is to execute quickly, the implementation uses several heuristics that trade precision for speed:

- When analyzing a method, JFTA only follows method calls up to a certain (configurable) depth. Thus, the analysis may miss method calls which potentially causes false positives or negatives.

- Information of different paths through a method is merged together. Thus, the analysis is path insensitive, which may cause false positives.

- Whenever the analysis is unable to determine the state of an object, it simply assumes that the object can be in any possible state. This may again generate false positives.

Due to the above heuristics, our approach is less precise than other tools such as [11]. However, in our setting we are interested on the effect of using enriched specifications rather than on absolute precision; and our results thus are likely to generalize to all sorts of typestate verifiers. This is further discussed in Section 4.5.

4.4.3 Results: True Positives

Table 5 summarizes the results for all changes that trigger exceptions. The next six columns list results using initial, enriched and complete models where available. "Reported" lists the number of defects for which the verifier reports a violation. "Actual" gives the number of cases where the reported method call exactly matches the call that raises the exception. For all numbers of reported errors, higher values are better.

The results show that, when using enriched models, the verifier pinpoints more violations than with initial versions. For the first three subjects, initial models cannot point to defects since they do not contain exceptional transitions. For the remaining three subjects, initial models also detect violations. For `signature` and `socket`, enriched models detect considerably more violations. For `zip`, both versions report violations for the same number of changes. However, enriched models more frequently point to the method call that raises the exception.

Better performance of enriched models in finding violations comes as no big surprise, as they include many more exceptional transitions than initial models. Still, the increase is considerable and the difference is statistically significant according to a paired-t-test with $p = 0.05$.

Table 4: Details about where tests came from and which applications where tested.

Subject	Test Source	# Tests	Application
javamail	Regression test suite	6	JVerify Binary Verifier (sourceforge.net)
javax.security	Regression test suite	5	Apache Jackrabbit (apache.org)
ristretto	Regression test suite	5	Fin J2EE calendar server (dev.java.net)
signature	Harmony compliance tests	16	opensc project (opensc-project.org)
socket	Harmony compliance tests	5	CRSMail Server (sourceforge.net)
zip	Harmony compliance tests	9	Huf 3.0 (sourceforge.net)

Table 5: Enriched models show more true positives.

		Initial model		Enriched model		Complete model	
Subject	Defects	Reported	Actual	Reported	Actual	Reported	Actual
javamail	5	0	0	4	3	n/a	n/a
javax.security	3	0	0	2	1	n/a	n/a
ristretto	28	0	0	25	15	21	19
signature	12	6	4	12	10	12	10
socket	49	2	2	48	47	n/a	n/a
zip	23	19	14	19	18	22	19

"Reported": number of defects detected;
"Actual": number of defects detected at the right method call.

Enriched models are better suited to finding errors than initial models.

For `zip` and `signature`, complete models find slightly more actual defects than enriched models. Thus for those two cases, models enriched by TAUTOKO are almost as good as manually created specifications. However, for `ristretto` complete models find 4 more defects (19 compared to 15). This is due to the nature of the typestate miner, which relies on the values of fields to capture an object's state (see Section 4.3). Even when using complete models, the verifier does not catch all defects. This is due to technical limitations of JFTA, such as the limited call stack depth.

Automatically enriched models can be almost as good as manually specified models.

4.4.4 *Results: False Positives*

Apart from finding errors, we would also like to have as few false positives as possible. To investigate the false positive rate of initial and enriched models, we repeated the above experiment with *changes that did not cause exceptions.*[4] For those changes, the verifier should not output violations.

The results of this experiment are shown in Table 6. The columns "Initial" and "Enriched" list the number of false positives for all types of models. For `javamail` and `signature`, we observe significantly fewer false positives. For the remaining subjects, the difference is smaller, but enriched models generally produce fewer false positives. A paired-t-test yields a p-value of 0.0124, which tells us that enriched models produce statistically significantly fewer false positives than initial models.

Table 6: Enriched models show fewer false positives.

Subject	Changes	Initial model	Enriched model	Complete model
javamail	28	26	2	n/a
javax.security	4	4	2	n/a
ristretto	53	53	47	7
signature	29	12	0	0
socket	460	300	283	n/a
zip	30	26	18	15

Enriched models produce fewer false positives than initial models.

Using complete models again yields the biggest improvement for `ristretto` with only seven false positives remaining. For the other two subjects, using manually created models provides no benefits over using enriched models from TAUTOKO.

4.5 Threats to Validity

As any empirical study, the results of our experiments are subject to threats to validity. We distinguish between threats to internal, external, and construct validity:

Threats to *external validity* concern our ability to generalize the results of our study. We cannot claim that the results of our experimental evaluation are generalizable. Our sample size is small; in total we investigate six subjects in twelve different applications. Also, our choice of subjects is biased towards more complex classes of projects with executable regression test suites. Less complex classes tend to generate only trivial models, and therefore TAUTOKO is unlikely to enrich them. However, applying TAUTOKO on such classes would not cause any harm, since the enriched

[4] We used coverage analysis to make sure that each change is actually covered.

model always contains the initial model. In practice, though, only specifications for classes that are complex enough to be misused should be distributed.

Threats to *internal validity* concern our ability to draw conclusions about the connections between our independent and dependent variables. Our process of manually creating complete models in Section 4.3 may be subject to errors or bias. When creating the models, we may have unintentionally left out states or transitions, which may influence our results. We therefore have used test cases to distinguish ambiguities wherever necessary. In addition, we make the models available at our website so that other researchers can investigate them (see Section 6).

Threats to *construct validity* concern the adequacy of our measures for capturing dependent variables. The last experiment uses our typestate verifier to compare models in terms of their ability to detect errors. A potential problem exists because the typestate verifier may miss violations due to over-approximations or technical limitations. We may therefore be unable to measure the number of correctly identified violations for a specification. However, our evaluation uses the same set of changes for both types of models. If over-approximations prevent the verifier from detecting a violation, it will do so for both types. As our evaluation focuses on the increase (or decrease for false positives), we believe that this is no real threat for the results of this experiment.

5. RELATED WORK

The idea of combining test case generation with specification mining was conceived by Xie and Notkin [32]. They present a generic *feedback loop* framework where specifications are fed into a test case generator, the generated tests are used to refine the specifications, and the refined specifications are again given as input to the test case generator. We extend this work by providing an implementation of the framework for typestate mining, as well as an evaluation of how useful enriched specifications are for a real-world application.

TAUTOKO uses techniques from several different areas of software engineering. The following sections summarize related work in the fields of test case generation, typestate verification, and specification mining.

5.1 Test Case Generation

There is a large body of work on test case generation, which is why we will limit the discussion to only a few representative approaches. If available, we cite surveys that provide more details in specific areas.

Several approaches use simple *randomized* algorithms to generate tests. Ciupa et al. [5] apply random testing to several industrial sized applications. Their work uses the AUTOTEST approach, which relies on invariants as test oracles. Milicevic et al. [23] present KORAT, which also leverages preconditions but works for JAVA programs. In contrast to random techniques, TAUTOKO specifically generates test cases to enrich a given initial model.

Another area in test case generation are *search-based* techniques. The majority of these approaches systematically analyze control-flow. Symbolic execution [17] simulates execution of the program using symbolic values rather than concrete ones and relies on constraint solvers to derive test data. Recently (e.g. [19]), combinations of concrete and symbolic execution were proposed to overcome limitations of symbolic execution in terms of scalability. A survey of existing search-based approaches can be found in [20]. In contrast to these approaches, TAUTOKO mutates the program to explore new behavior, thus changing the control flow rather than analyzing it.

TAUTOKO is an instance of a *model-based* test generation tool. Such tools require the presence of a model that describes the intended system behavior. This model is then used to derive tests or input data. They come in very different forms, e.g. as finite state machines, or algebraic specifications. A survey on existing model-based approaches can be found in [15]. On example of a model-based testing tool is SPECEXPLORER [29], which is developed by Microsoft Research. SPECEXPLORER explores specifications written in SPEC# [2] model-checking techniques and provides test cases for explored behavior.

The idea of mutating the test suite to generate test cases was inspired by work of Tonella [28]. He proposes *evolutionary testing:* using genetic algorithms, an initially generated test suite is mutated to satisfy a given coverage criterion. In contrast to evolutionary testing, TAUTOKO uses a model to guide test case generation. To our knowledge, we are the first that use test-generation techniques to improve the quality of mined typestates.

In the area of web application testing, Mesbah et al. [21] extract state machines that describe the user interface of AJAX applications. Their tool called ATUSA derives sequences of operations that are executed to explore the application and trigger defects. In contrast, our approach explores JAVA classes and generates new tests to enrich specifications.

Gupta and Heidepriem [14] explore a new structural coverage criterion based on dynamic invariants. They use DAIKON [10] to mine an initial set of likely invariants. Based on this set, Gupta and Heidepriem generate a new test suite that tries to cover as many invariants as possible. This test suite can be used to remove spurious invariants from the initial set. In contrast, TAUTOKO mines typestate automata and uses mutation to generate new tests.

5.2 Typestate Verification

The term typestate was coined in 1986 by Strom and Yemini [27]. Initially, typestates were used to distinguish uninitialized and valid pointers. This information was used to detect potential null pointer dereferences and memory leaks in PASCAL programs.

Since then, several approaches have been developed for different platforms such as .NET [9] or JAVA [12] with varying levels of precision. A promising sound typestate verifier for JAVA was presented by Fink et al. [11]. The tool uses a staged approach with a total of four stages: early stages use imprecise and fast techniques to filter instances that need not be considered in later (more precise and thus expensive) stages. The last stage is only required for objects referenced by more than one method or objects stored in collections. Fink et al. report analysis times ranging from one to ten minutes for projects with up to 200 classes. In contrast to their approach, JFTA is less precise due to its conservative handling of arrays and collections. We would expect that using the tool by Fink et al. would further reduce the number of false positives in our evaluation.

5.3 Specification Mining

The large body of work on mining specifications can be grouped into dynamic and static approaches. The first technique by Cook and Wolf [6] considers the general problem of extracting a finite state machine based model from an event trace. They reduce the problem to the well-known grammar inference problem [13] and discuss algorithmic, statistical and hybrid approaches. Later, Larus et al. [1] proposed mining specifications for automatic verification. Their approach learns probabilistic finite state automata for C programs. Following the assumption that common behavior is correct behavior, Larus et al. use the inferred automata to search for anomalies in other executions of the program.

Among the first approaches that specifically mine models for classes is the work by Whaley et al. [31]. Their technique mines models with anonymous states and slices models by grouping methods that access the same fields. Lorenzoli et al. [18] mine so-called extended finite state machines with anonymous states. To compress models, the gk-tail algorithm merges states that have the same k-future.

In terms of static techniques, there is also a huge number of different approaches. Wasylkowski et al. [30] mine object usage models that describe the usage of an object in a program. They apply concept analysis to find code locations where rules derived from usage models are violated. Ramanathan et al. [24] use an inter-procedural path-sensitive analysis to infer preconditions for method invocations. Shoham et al. [25] discover that static mining of automata based specifications requires precise aliasing information to produce reliable results.

In the area of web services, Bertolino et al. [3] mine behavior protocols that describe the usage of a web service. The approach uses a sequence of synthesis and testing stages that uses heuristics to refine an initially mined automaton. In contrast, TAUTOKO mines typestate automata for JAVA programs.

6. 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 prototype explores previously unobserved aspects of the execution space. The resulting enriched 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, and significantly fewer false positives. Generally, we expect test case generation to be applicable to all techniques of dynamic specification mining, improving the effectiveness of mined specifications.

Despite these advances, there is much more to do. Besides general improvements regarding performance and scalability, our future work will focus on the following topics:

Technical Improvements. TAUTOKO can only apply one mutation at a time, which is why some states cannot be fully explored. This is due to limitations of the instrumentation framework. In the future, we would like to extend TAUTOKO such that arbitrary combinations of mutations are possible.

Test Case Generation. TAUTOKO's strategy for generating tests is simple but effective and only requires a test suite as input. However, there are many other approaches to test case generation that could be used just as well. One idea for future work is to compare different strategies in terms of their ability to enrich a specification.

A Multi-Faceted View of Program Analysis. As demonstrated in this paper, there are many ways to infer program properties: Not only can we examine their code (static analysis) or their executions (dynamic analysis); but 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.

All components of TAUTOKO are available for download. We also include all models generated for our evaluation subjects, as well as manually created models for three classes. To learn more about TAUTOKO, visit its Web site:

http://www.st.cs.uni-saarland.de/models/

Acknowledgments. This work is funded by Deutsche Forschungsgemeinschaft, Ze509/4-1 and Hasler-Stiftung, Grant no. 2327. We thank the anonymous reviewers for their help in improving the final version of this paper. Klaas Boesche provided lots of support implementing JFTA. Andrzej Wasylkowski, David Schuler, Yana Mileva and Gordon Fraser provided helpful comments on earlier revisions of this paper.

7. REFERENCES

[1] G. Ammons, R. Bodík, and J. Larus. Mining specifications. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, Jan. 16–18, 2002.

[2] M. Barnett, R. Deline, M. Fähndrich, B. Jacobs, K. R. Leino, W. Schulte, and H. Venter. The Spec# programming system: Challenges and directions. pages 144–152, 2008.

[3] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering Conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 141–150, New York, NY, USA, 2009. ACM.

[4] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-oriented programming systems and applications*, pages 301–320, New York, NY, USA, 2007. ACM.

[5] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *ISSTA '07: Proceedings of the 2007 International symposium on Software testing and analysis*, pages 84–94, New York, NY, USA, 2007. ACM.

[6] J. Cook and A. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[7] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA 2006: ICSE Workshop on Dynamic Analysis*, May 2006.

[8] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, November 2009.

[9] R. Deline and M. Fähndrich. Typestates for objects. In *In Proc. 18th ECOOP*, pages 465–490. Springer, 2004.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[11] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions Software Engineering Methodology*, 17(2):1–34, 2008.

[12] E. Geay, E. Yahav, and S. Fink. Continuous code-quality assurance with SAFE. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 145–149, New York, NY, USA, 2006. ACM Press.

[13] E. Gold. Language identification in the limit. *Information and Control*, pages 447–474, 1967.

[14] N. Gupta and Z. V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. *Automated Software Engineering, International Conference on*, 0:49, 2003.

[15] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, and G. Lüttgen. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76, 2009.

[16] http://ostatic.com/ristretto. Ristretto 1.0, 2010.

[17] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[18] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th International Conference on Software engineering*, pages 501–510, New York, NY, USA, 2008. ACM.

[19] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[20] P. Mcminn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[21] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[22] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *Computer*, 42:46–55, 2009.

[23] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 771–774, Washington, DC, USA, 2007. IEEE Computer Society.

[24] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming language design and implementation*, pages 123–134, New York, 2007. ACM.

[25] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA '07: of the 2007 int. symposium on Software testing and analysis*, pages 174–184, New York, 2007. ACM.

[26] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Not.*, 41(6):387–400, 2006.

[27] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *Transactions Software Engineering*, 12(1):157–171, 1986.

[28] P. Tonella. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes*, 29(4):119–128, 2004.

[29] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. *SIGSOFT Software Engineering Notes*, 30(5):273–282, 2005.

[30] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering Conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 35–44, New York, NY, USA, 2007. ACM.

[31] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International symposium on Software testing and analysis*, pages 218–228, New York, NY, USA, 2002.

[32] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES 03)*, volume 2931 of *LNCS*, pages 60–69, October 2003.