11-2013

TzuYu: Learning stateful typestates

Hao XIAO

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Shang-Wei LIN

Chengnian SUN

Citation

XIAO, Hao; SUN, Jun; LIU, Yang; LIN, Shang-Wei; and SUN, Chengnian. TzuYu: Learning stateful typestates. (2013). *Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, USA, November 11-15*. 432-442.
Available at: https://ink.library.smu.edu.sg/sis_research/5007

TzuYu: Learning Stateful Typestates

Hao Xiao*, Jun Sun†, Yang Liu*, Shang-Wei Lin‡ and Chengnian Sun§
*School of Computer Engineering, Nanyang Technological University
†Singapore University of Technology and Design
‡Temasek Laboratories, National University of Singapore
§School of Computing, National University of Singapore

***Abstract*—Behavioral models are useful for various software engineering tasks. They are, however, often missing in practice. Thus, specification mining was proposed to tackle this problem. Existing work either focuses on learning simple behavioral models such as finite-state automata, or relies on techniques (e.g., symbolic execution) to infer finite-state machines equipped with data states, referred to as *stateful typestates*. The former is often inadequate as finite-state automata lack expressiveness in capturing behaviors of data-rich programs, whereas the latter is often not scalable. In this work, we propose a fully automated approach to learn stateful typestates by extending the classic active learning process to generate transition guards (i.e., propositions on data states). The proposed approach has been implemented in a tool called TzuYu and evaluated against a number of Java classes. The evaluation results show that TzuYu is capable of learning correct stateful typestates more efficiently.**

I. Introduction

Behavioral models or specifications are useful for various software engineering tasks. For instance, (object) typestates [11], [13], [27], [34] are important for program debugging and verification. A precise (and preferably concise) typestate is useful for understanding third-party programs. In practice, however, such models are often inadequate and incomplete. To overcome this problem, learning based specification mining [5] was proposed to automatically generate behavioral models from various software artifacts, e.g., source code [2], execution traces [29] and natural language API documentation [37]. This approach is promising as it requires no extra user efforts.

Existing approaches on learning typestates (also known as interface specification [4]) can be broadly categorized into two groups. One focuses on learning behavioral models in the forms of finite-state automata, without data states. These methods are often inadequate in practice, as it is known that finite-state automata lack expressiveness in modeling data-rich programs. Consider a simple example of a *Stack* class with two operations: *push* and *pop*. A typestate of the *Stack* should specify the following language: the number of *push* operations in any valid trace of the model must be no less than the number of *pop* operations. It is known that this language is irregular and therefore beyond the expressiveness of finite-state automata. On the other hand, the model of the *Stack* can be easily expressed using a finite-state machine with a guard condition on the *pop* operation: $size \geq 1$ where *size* denotes the number of items in the stack. The central issue is thus: how to identify the proposition $size \geq 1$ systematically and automatically.

The other group learns stateful typestates using relatively heavy-weight techniques like SMT/SAT solving. For instance, Alur *et al.* [4] propose to synthesize interface specifications for Java classes based on predicate abstraction, which relies on theorem proving. Similarly, Giannakopoulou *et al.* [15] propose to learn typestates through symbolic execution (which relies on SMT solving) and refinement. Given that existing theorem proving and SMT/SAT techniques are still limited in handling complicated data structures and control flows, these methods are often limited to small programs.

In this paper, we propose an alternative approach to learning stateful typestates from Java programs. The key idea is to extend an active learning algorithm with an approach to automatically learning transition guards (i.e., propositions on data states). Our approach takes the source code of a class as the only input and generates a stateful typestate through a series of testing, learning and refinement. Fig. 1 shows the high level architecture of our approach. There are three main components. The learner constructs a typestate based on the L* algorithm [6]. It drives the learning process by generating two kinds of queries. One is the membership query, i.e., whether a sequence of events (i.e., a trace) of the current typestate is valid. The other is the candidate query, i.e., whether a candidate typestate matches the 'actual' typestate. The tester acts as a teacher in the classic active learning setting. It takes queries from the learner and responds accordingly based on testing results. In the original L* algorithm, the model to be learned is a finite-state automaton and a trace can be either valid or invalid but never both. However, in our setting, it is possible that two executions have the same sequences of method calls on the same object but lead to different outcomes (i.e., error or no-error), due to different inputs to the method calls (which in turn result in different data states). In such a case, alphabet refinement is performed, by splitting one event into multiple events, each of which has a different guard condition so that the traces are distinguished. The refiner is used to automatically identify proper guard conditions. In the following, we use a simple example to illustrate how our method works.

We take the *java.util.Stack* class in Java (SE 1.4.2) as the running example. Without loss of generality, let us focus on the following two methods: *push* (which takes an object as an input) and *pop*, and one data field *eleCount* (inherited from the *java.util.Vector* class) which denotes the number of elements in the stack. Initially, we have an alphabet containing

978-1-4799-0215-6/13 © 2013 IEEE

ASE 2013, Palo Alto, USA

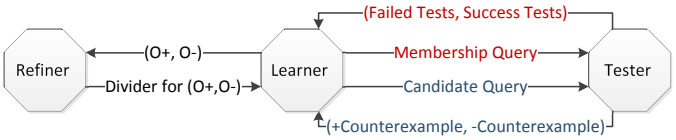
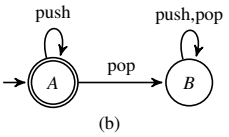


Fig. 1. The high-level architecture of TzuYu.

	$\langle\rangle$
$\langle\rangle$	1
$\langle push \rangle$	1
$\langle pop \rangle$	0
$\langle pop, push \rangle$	0
$\langle pop, pop \rangle$	0

(a)

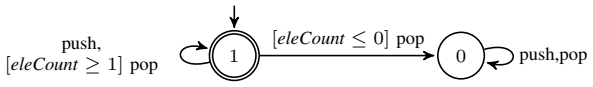


(b)

Fig. 2. The first observation table (a) and candidate Typestate (b).

	$\langle\rangle$
$\langle\rangle$	1
$\langle push \rangle$	1
$\langle [!(eleCount \geq 1)]pop \rangle$	0
$\langle [eleCount \geq 1]pop \rangle$	1
$\langle [!(eleCount \geq 1)]pop, push \rangle$	0
$\langle [!(eleCount \geq 1)]pop, [eleCount \geq 1]pop \rangle$	0
$\langle [!(eleCount \geq 1)]pop, [!(eleCount \geq 1)]pop \rangle$	0

(a)



(b)

Fig. 3. The second observation table (a) and candidate Typestate (b) generated by TzuYu.

two events corresponding to the two methods. Given an instance of the Stack class, the learner generates a number of membership queries, i.e., a sequence of method calls. Given one membership query, the tester generates multiple test cases which have the same sequence of method calls (with different arguments) and answers the query. The queries and testing results are summarized in the observation table (refer to details in Section II-B), as shown in Fig. 2 (a) where $\langle\rangle$ is an empty sequence of method calls; and $\langle pop, push \rangle$ denotes the sequence of calling *push* after *pop*. The 0s in column $\langle\rangle$ denote that all tests generated for the sequence $\langle pop \rangle$ and then $\langle\rangle$ result in an exception or assertion failure (hereafter failure). The 1s denote that none of the tests result in failure. Based on the observation table, the learner generates a candidate typestate as presented in Fig. 2 (b). Note that the typestate is a finite-state automaton with one accepting state, i.e., state *A*.

Next, the learner asks a candidate query, i.e., is the typestate in Fig. 2 (b) the right typestate? The tester takes the candidate typestate and performs random walking, i.e., randomly generates a set of tests which correspond to traces of the typestate. Notice that a trace of the typestate is either accepting (i.e., ending with an accepting state) or otherwise. Through the random walking, the tester identifies one inconsistency between the typestate and the class under analysis. That is, the typestate predicates that calling *pop* from state *A* always results in failure, whereas it is not always the case. For instance, calling method *push* first (which leads to state *A*) and then *pop* results in no failure.

The existence of inconsistency suggests that the typestate must be refined. We collect data states of the stack at state *A* before calling method *pop* and partition them into two sets, i.e., ones which lead to failure after invoking *pop* and the rest. Next, the refiner is consulted to generate a proposition ϕ such that all data objects in the first set satisfy ϕ while all the rest violate ϕ. The technique used by the refiner is based on Support Vector Machines (SVMs) [31]. In the above example, the generated proposition is $eleCount \geq 1$. Next, we re-start the learning process with an alphabet which contains three events: *push*, $[eleCount \geq 1]pop$, and $[!(eleCount \geq 1)]pop$ where $[eleCount \geq 1]pop$ denotes the event of calling *pop* when the condition $eleCount \geq 1$ is satisfied. After a series of membership queries, the learner constructs the observation table as shown in Fig. 3 (a).

Notice that all tests corresponding to $[eleCount \geq 1]pop$ result in no failure and therefore it is marked 1 in the table. A new candidate typestate is then generated from the table, as shown in Fig. 3 (b). The tester performs random walking again and finds no inconsistency. We then present Fig. 3 (b) as the resultant typestate after some simple bookkeeping on Fig. 3 (b) (by transforming $!(eleCount \geq 1)$ to $eleCount \leq 0$ using the fact that *eleCount* is an integer). □

The novelty of our approach is on integrating a refiner into the active learning process so as to learn typestates for data-rich programs. In particular, by adopting techniques from machine learning community, we are able to automatically generate propositions for alphabet refinement. The refiner acts as an abstract mapper between the learner and the class under analysis. Compared with existing techniques on finding the right proposition (e.g., [15]), our approach improves the performance of typestates generation as it avoids SMT/SAT encoding and solving. Furthermore, to learn concise stateful typestates efficiently, we investigate the interplay between learning and refinement and develop an algorithm which avoids re-starting learning when alphabet refinement occurs. The method has been implemented in a tool named TzuYu[1] and our experiments show that TzuYu is able to learn meaningful and concise typestates efficiently.

The remainder of the paper is organized as follows. Section II presents preliminary introduction to the concepts and techniques used in our approach. Section III presents the details of our approach. Section IV presents details on the implementation of TzuYu and Section V evaluates its performance with experiments. Section VI discusses related work. Section VII concludes the paper.

II. Preliminaries

In this section we formalize the definitions related to stateful typestate and introduce the techniques used in our approach.

A. Definitions

The input to our method is a Java class (e.g., the Stack class) which is constituted by a set of instance variables (which could

[1]TzuYu is commonly known as the best student of Confucius.

be objects of other classes) and methods. In this work, we fix one object of the given class as the main receiver and inspect behaviors of all instances of the class through this object. An object state is the status of the object, i.e., the valuation of its variables. For each object, there is an initial object state[2], i.e., the initial valuation of the variables. A method is a function which takes one object state and returns a new one. A concrete execution *ex* of an object is a finite sequence

$$ex = \langle o_0, m_0(\overrightarrow{p_0}), o_1, m_1(\overrightarrow{p_1}), \cdots, o_k, m_k(\overrightarrow{p_k}), o_{k+1} \rangle$$

where o_i is an object state and $m_i(\overrightarrow{p_i})$ is a method call with concrete arguments $\overrightarrow{p_i}$. A failed execution is an execution which results in an exception or assertion failure. A successful execution is one which does not fail.

The output of our method is a stateful typestate, which is defined on top of the deterministic finite-state automaton.

Definition 1: A deterministic finite-state automaton (hereafter DFA) is a tuple $\mathcal{D} = (S, \Sigma, init, \rightarrow, F)$ such that S is a finite set of states; $init \in S$ is an initial state; Σ is the alphabet which is a finite set of events; $\rightarrow: S \times \Sigma \rightarrow S$ is a transition function and $F \subseteq S$ is a set of accepting states. □

A trace of $\mathcal{D}$ is a sequence $tr = \langle s_0, e_0, s_1, \cdots, s_n, e_n, s_{n+1} \rangle$ such that $s_0 = init$ and $(s_i, e_i, s_{i+1}) \in \rightarrow$ for all i. *tr* is accepting if $s_{n+1} \in F$. Otherwise, it is non-accepting. The language of $\mathcal{D}$ is the set of all accepting traces of $\mathcal{D}$. In an abuse of notations, we write $s \xrightarrow{tr} s'$ to denote that trace *tr* from state s leads to state s' and write $tr(s)$ to denote s'. For two traces tr_0 and tr_1, we write $tr_0 \cdot tr_1$ to denote their concatenation.

Definition 2: A (stateful) typestate of a Java class is a tuple $\mathcal{T} = (Prop, Meth, \mathcal{D})$ such that *Prop* is a set of propositions, which are Boolean expressions over variables in the class; *Meth* is the set of method names in the class; $\mathcal{D} = (S, \Sigma, init, \rightarrow, F)$ is a DFA such that $\Sigma \subseteq Prop \times Meth$. □

In the Stack example, a proposition in *Prop* can be constituted by *eleCount*, *capacity* (inherited from Vector), any data field of *elementData* (e.g., *elementData.length*), etc. Set *Meth* contains *push* and *pop*. By definition, typestates are deterministic in this work. Notice that an event in Σ is a pair, i.e., a guard condition g in *Prop* and a method name e in *Meth*. For brevity, a transition is written as $(s, [g]e, s')$. A typestate abstracts all executions of an object of the class. In particular, a trace $tr = \langle s_0, [g_0]e_0, s_1, [g_1]e_1, s_2, \cdots, s_n, [g_n]e_n, s_{n+1} \rangle$ is an abstraction of the execution *ex* above if they have the same sequence of methods (i.e., $e_i = m_i$ for all i) and all the guard conditions are satisfied (i.e., g_i is satisfied by o_i and method arguments $\overrightarrow{p_i}$ for all i). We denote the set of concrete executions of *tr* as $con(tr)$. Given an execution *ex* and an alphabet Σ, we can obtain the corresponding trace, denoted as $abs(ex)$, by testing which proposition in *Prop* is satisfied for each method call in *ex*.

A typestate $\mathcal{D}$ is said to be safe (or sound), if for every accepting trace *tr* of $\mathcal{D}$, every execution in $con(tr)$ is successful. It is complete if for every concrete execution *ex* of the class, there is an accepting trace *tr* such that $ex \in con(tr)$.

[2]For brevity, a constructor is treated in the same way as a normal method except that it must be called initially and calling it later leads to failure.

B. The L* Algorithm

The learner extends the original L* algorithm [6] with lazy alphabet refinement, which is introduced later in section III-C. In the following we introduce the original L* algorithm.

The L* algorithm assumes that the system to be learned $\mathcal{D}$ is in the form of DFA with a fixed alphabet Σ and learns a DFA with the minimal number of states that accepts the same language of $\mathcal{D}$. During the learning process, the L* algorithm interacts with a *Minimal Adequate Teacher* (teacher for short) by asking two types of queries: membership queries and candidate queries. A *membership query* asks whether a trace *tr* is a trace of $\mathcal{D}$, whereas a *candidate query* asks whether a DFA $\mathcal{C}$ is equivalent to $\mathcal{D}$, i.e., $\mathcal{C}$ and $\mathcal{D}$ have the same language.

During the learning process, the L* algorithm stores the membership query results in an *observation table* (P, E, T) where $P \subseteq \Sigma^*$ is a set of prefixes; $E \subseteq \Sigma^*$ is a set of suffixes; and T is a mapping function such that $T(tr, tr') = 1$ if *tr* is a trace in P or a trace in P attached with an event in Σ; and tr' is a trace is E and $tr \cdot tr'$ is a trace of the system; otherwise, $T(tr, tr') = 0$. In the observation table, the L* algorithm categorizes traces based on Myhill-Nerode Congruence [17].

Definition 3: We say two traces *tr* and tr' are *equivalent*, denoted by $tr \equiv tr'$, if $tr \cdot \rho$ is a trace of $\mathcal{S}$ iff $tr' \cdot \rho$ is a trace of $\mathcal{S}$, for all $\rho \in \Sigma^*$. Under the equivalence relation, we can say *tr* and tr' are the *representing trace* of each other with respect to $\mathcal{S}$, denoted by $tr = [tr']_r$ and $tr' = [tr]_r$. □

The L* algorithm always tries to make the observation table *closed* and *consistent* with membership queries. An observation table is *closed* if for all $tr \in P$ and $e \in \Sigma$, there always exists $tr' \in P$ such that $tr \cdot \langle e \rangle \equiv tr'$. An observation table is *consistent* if for every two elements $tr, tr' \in P$ such that $tr \equiv tr'$, then $(tr \cdot \langle e \rangle) \equiv (tr' \cdot \langle e \rangle)$ for all $e \in \Sigma$. If the observation table (P, E, T) is closed and consistent, the L* algorithm constructs a corresponding candidate DFA $\mathcal{C} = (S, init, \Sigma, \rightarrow, F)$ such that

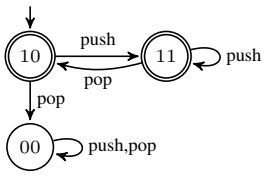
- S contains one state for each trace in P; notice that equivalent traces in P correspond to the same state.
- *init* is the state corresponding to the empty trace $\langle\rangle$;
- for any state s in S which corresponds to a trace *tr* and $e \in \Sigma$, $(s, e, s') \in \rightarrow$, where s' is the state for the trace $[tr \cdot \langle e \rangle]_r$ in P;
- a state s is in F iff the corresponding trace *tr* satisfies $T(tr) = 1$.

Subsequently, L* raises a candidate query on whether $\mathcal{C}$ is equivalent to the system to be learned.

If $\mathcal{C}$ is equivalent to the system, $\mathcal{C}$ is returned as the learning result. Otherwise, the teacher identifies a counterexample, say *tr*, which is then analyzed to find a *witness suffix*. A witness suffix is a trace that, when appended to the two traces, provides enough evidence for the two traces to be classified into two equivalence classes under the Myhill-Nerode Congruence. Let

	$\langle\rangle$	$\langle pop\rangle$
$\langle\rangle$	1	0
$\langle push\rangle$	1	1
$\langle pop\rangle$	0	0
$\langle pop, push\rangle$	0	0
$\langle pop, pop\rangle$	0	0
$\langle push, push\rangle$	1	1
$\langle push, pop\rangle$	1	0

(a)



(b)

Fig. 4. The second observation table (a) candidate DFA (b) generated by the classic L* algorithm.

tr be the concatenation of two traces tr_0 and tr_1, i.e., $tr_0 \cdot tr_1 = tr$. Let *s* be the state reached from state *init* via trace tr_0, i.e., $init \xrightarrow{tr_0} s$. tr_1 is the *witness suffix* of *tr*, denoted by $WS(tr)$, if $s \xrightarrow{tr_1} s'$ and $s' \neq \mathcal{D}(tr)$, where $\mathcal{D}(tr)$ denotes the state reached after running *tr* on $\mathcal{D}$. Once the witness suffix $WS(\sigma_{ce})$ is obtained, L* uses $WS(\sigma_{ce})$ to refine the candidate DFA $\mathcal{C}$ until $\mathcal{C}$ is equivalent to the system. We refer readers to the work of Lin *et al.* [22], [23] for more details of L* Algorithm with examples.

Angluin [6] proved that as long as the unknown language *U* is regular, the L* algorithm will learn an equivalent minimal DFA with at most $n-1$ candidate queries and $O(|\Sigma| n^2 + n \log m)$ membership queries, where *m* is the length of the longest counterexample returned by the teacher and *n* is the number of states of the minimal DFA.

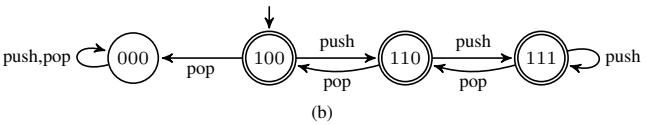
Example 1: We again use the Stack example to illustrate how L* works and also why it does *not* work when the target class cannot be captured by a DFA. After a series of membership queries, L* constructs the first candidate DFA, as shown in Fig. 2 (b), and performs a candidate query for the DFA. The teacher answers "no" with a positive counterexample $\langle push, pop\rangle$, which should be included into the behavior of the candidate. After analyzing the counterexample, the witness suffix $\langle pop\rangle$ is added into the set of suffixes *E* of the observation table, and the closed observation table is shown in Fig. 4 (a). Based on the observation table, L* constructs the second candidate DFA, as shown in Fig. 4 (b), and performs a candidate query for the candidate. The teacher answers "no" again with another positive counterexample $\langle push, push, pop, pop\rangle$. This time, the witness suffix $\langle pop, pop\rangle$ is added into the set of suffixes *E* of the observation table, and the closed observation table is shown in Fig. 5 (a). Based on the observation table, L* constructs the third candidate DFA, as shown in Fig. 5 (b), and performs a candidate query for the third one. The reader may find that after the *i*th candidate query for $i \in \mathbb{N}$, there is always a witness suffix $\langle (pop)^i\rangle$ showing that the candidate DFA is incorrect, and one additional state will be added to the candidate DFA, which makes the L* learning process non-terminating. □

III. Detailed Approach

In this section we first introduce the detailed design of the tester and refiner and then introduce the learner which interacts with the tester and learner to learn the typestate.

	$\langle\rangle$	$\langle pop\rangle$	$\langle pop, pop\rangle$
$\langle\rangle$	1	0	0
$\langle push\rangle$	1	1	0
$\langle pop\rangle$	0	0	0
$\langle pop, push\rangle$	0	0	0
$\langle pop, pop\rangle$	0	0	0
$\langle push, push\rangle$	1	1	1
$\langle push, pop\rangle$	1	0	0
$\langle push, push, push\rangle$	1	1	1
$\langle push, push, pop\rangle$	1	1	0

(a)



(b)

Fig. 5. The third observation table (a) and candidate DFA (b) generated by the classic L* algorithm.

A. The Tester

The tester acts as the teacher for L* algorithm. Ideally, given a membership query for a trace *tr*, the teacher should answer either yes or no. Since *tr* can be mapped into a set of concrete executions $con(tr)$, that is to say that the teacher should answer yes iff all executions in $con(tr)$ are successful and answer no iff all executions in $con(tr)$ are failed. Similarly, given a candidate query, the tester should answer yes iff the candidate typestate is safe and complete.

Having a perfect teacher in our setting is infeasible for two main reasons. Firstly, the set $con(tr)$ is infinite (with different arguments for method calls) in general and hence checking whether all executions in $con(tr)$ are successful or not is highly non-trivial. Secondly, it could be that some executions in $con(tr)$ are successful, whereas some are failed. For instance, assume the class given is *java.util.vector* and *tr* is $\langle addAll\rangle$. A concrete execution with a method call *addAll* and argument *null* results in exception, whereas a non-null argument results in success. We tackle the former problem by using guided random testing as the teacher, as we discuss below. The latter problem is solved by alphabet refinement, as we show in Section III-B.

In the following, we show how the tester is used as a teacher for membership queries and candidate queries. Given a membership query *tr* as follows:

$$tr = \langle s_0, [g_0]m_0, s_1, [g_1]m_1, s_2, \cdots, s_n, [g_n]m_n, s_{n+1}\rangle$$

the tester's task is to identify multiple concrete executions as follows: $\langle o_0, m_1(\overrightarrow{p_1}), o_2, m_2(\overrightarrow{p_2}), \cdots, o_k, m_k(\overrightarrow{p_k}), o_{k+1}\rangle$. In other words, to automatically generate the arguments for all method calls such that all guard conditions g_i are satisfied. This task is in general highly non-trivial and requires techniques like SAT/SMT solving. In the name of scalability, we instead apply testing techniques for argument generation. In particular, the approach of Randoop [28] is adopted. In the following, we briefly introduce the idea and refer readers to details in [28].

Given *tr*, we generate arguments for each method call one-by-one in sequence. Given a typed parameter, the idea is to randomly generate a value from a pool of type-compatible values. This pool composes of a set of pre-defined value (e.g.,

a random integer for an integer type, *null* or an object with the default object state for a user-defined class, etc.) but also type-compatible objects that have been generated during the testing process. We remark that in order to re-create the same object, we associate each object with the execution which produces the object state. Given one value for each parameter, we then evaluate whether g_i is true or not. If g_i is true, we proceed with next method call.

There are four possible outcomes of the random testing. If all tests are successful, the answer to the query is yes, i.e., *tr* should be an accepting trace. If all tests are failed, the answer is no, i.e., *tr* should be a non-accepting trace. If there are both successful tests and failed tests (for *tr* or a prefix of *tr*), the tests are passed to the refiner for alphabet refinement as we show later. Lastly, due to the limitation of random testing (i.e., the price we pay to avoid theorem proving), it is possible that some guard condition g_i is never satisfied by the generated arguments. In other words, we fail to find any concrete execution in $con(tr)$. In such a case, we optimistically answer yes so that the resultant typestate is more permissive.

To answer a candidate query with a typestate $\mathcal{C}$, we use random walk [9], [10], [21] to generate a suite of test cases. Note that the approach of Randoop [28] is again used. Test cases which are inconsistent with the typestates are collected into two sets: positive counterexamples and negative counterexamples. A positive counterexample is a successful test whose corresponding trace *tr* is non-accepting. A negative example is a failed test whose corresponding trace *tr* is accepting. If both sets are empty, we answer the query with a yes, i.e., the typestate is the final output. If either of the two sets is not empty, the typestate is 'invalid' and a counterexample must be presented to the learner. In the original L* algorithm, presenting any of the counterexamples will do. It is however more complicated in our setting as we show below.

For each state s in the typestate $\mathcal{C}$, we identify a set of executions in the test suite which end at the state, denoted as E_s. For each $e \in \Sigma$, we extend each execution in E_s with a method call corresponding to e and obtain a new set denoted as E_s^e. If all of the executions result in failure whereas a transition labeled with e from s leads to an accepting state in $\mathcal{C}$, the tester reports that $\mathcal{C}$ is invalid and picks one execution in E_s^e and presents its corresponding abstract trace as a counterexample. Similarly, if all of the executions are successful, whereas a transition labeled with e from s leads to a non-accepting state, the tester presents a counterexample. Lastly, if some of the executions in E_s^e result in failure and others result in success, the refiner is consulted to perform alphabet refinement.

B. The Refiner

There are two different scenarios when the refiner is consulted. One is with a membership query *tr* and a set of tests in $con(tr)$ such that for some of the executions (denoted as T^-), performing the last method call (with the generated arguments) results in failure, whereas for the rest of the executions (denoted as T^+), performing the last call results in success. In this case, alphabet refinement is a must as all the tests have the same trace *tr* and therefore they cannot be distinguished without alphabet refinement.

Given an execution in T^- or T^+, we can obtain a data state pair $(o, \overrightarrow{p})$ where o is the object state of the main instance prior to the last method call and $\overrightarrow{p}$ is the list of arguments of the last method call. Let O^- be the set of all pairs we collect from executions in T^- and O^+ be the set of all pairs we collect from executions in T^+. Intuitively, there must be something different between O^- and O^+ such that T^- fails and T^+ succeeds. The refiner's job is to find a *divider*, in the form of a proposition, such that O^- and O^+ can be distinguished. Formally, a divider for O^+ and O^- is a proposition ϕ such that for all $o \in O^+$, o satisfies ϕ and for all $o' \in O^-$, and o' does not satisfy ϕ. From another point of view, there must be some invariant for all object states in O^+ (denoted as inv^+) and some invariant for all object states in O^- (denoted as inv^-) such that inv^+ implies ϕ and inv^- implies the negation of ϕ.

The refiner in our work is based on techniques developed by machine learning community, in particular, Support Vector Machines (SVMs) [31]. SVM is a supervised machine learning algorithm for classification and regression analysis. We use its binary classification functionality. Mathematically, the binary classification functionality of SVMs works as follows. Given two data states (say O^+ and O^-), each of which can be viewed as a vector of numerical values (e.g., floating-point numbers), it tries to find a separating hyperplane $\Sigma_{i=1}^n c_i * x_i = c$ such that (1) for every positive data state $(p_1, p_2, \cdots, p_n) \in O^+$ such that $\Sigma_{i=1}^n c_i * p_i > c$ and (2) for every negative data state $(m_1, m_2, \cdots, m_n) \in O^-$ such that $\Sigma_{i=1}^n c_i * m_i < c$. As long as O^+ and O^- are linear separable, SVM is guaranteed to find a separating hyperplane, even if the invariants inv^+ and inv^- may not be linear. Furthermore, there is usually more than one hyperplane that can separate O^+ from O^-. In this work, we choose the *optimal margin classifier* (see the definition in [33]) if possible. This separating hyperplane could be seen as the strongest witness why the two data states are different.

In order to use SVM to generate dividers, each element in O^+ or O^- must be casted into a vector of numerical types. In general, there are both numerical type (e.g., *int*) and categorical type (e.g., *String*) variables in Java programs. Thus, we need a systematic way of mapping arbitrary object states to numerical values so as to apply SVM techniques. Furthermore, the inverse mapping is also important to feed the SVM results back to the original program. Our approach is to systematically generate a *numerical value graph* from each object type and apply SVM techniques to values associated with nodes in the graph level-by-level. We illustrate our approach using an example in the following.

Fig. 6 shows part of the numerical value graph for type *Stack* (where many data fields have been omitted for readability). A rectangle (with round corners) represents a categorical type, whereas a circle associated with the type denotes a numerical value which can be extracted from the type. Notice that a categorical type is always associated with a Boolean type value which is true *iff* the object is null. An edge reads as "contains". For instance, a Stack type contains an object of

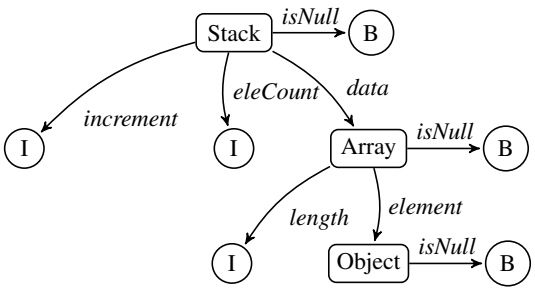


Fig. 6. The numerical value graph for Stack.

type "Array" (i.e., *elementData*), which in turn contains objects of type "Object". For readability, each edge is labeled with an abbreviated variable name and each node is labeled with the type. To obtain a vector of numerical values from a type, we traverse through the graph level-by-level to collect numerical values associated with each type. In general, the graph could be huge if a type contains many variables. For the purpose of typestate learning, however, it is often sufficient to look at only the top few levels.

In the following, we demonstrate how the graph is used. Assume the last event of the membership query is $[true]pop$ and the two sets of object states are O^+ and O^- prior to the method call. Given the receiver object of the method call is a Stack, the refiner first abstracts O^+ and O^- using level-0 numerical values in the graph, i.e., *isNull*, *eleCount* and *increment* which is the amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity, inherited from the Vector class. Next, the refiner tries to generate a divider which separates the abstracted O^+ from that of O^-. Assume that O^+ contains two object states and the abstracted O^+ is a set: $\{\langle 0,1,1\rangle, \langle 0,2,1\rangle\}$ where $\langle 0,1,1\rangle$ denotes a Stack object which is not null (i.e., 0 means that *isNull* is false), with *eleCount* being 1 and with *increment* being 1. Assume that the abstracted O^- is: $\{\langle 0,0,1\rangle, \langle 0,0,1\rangle\}$. SVM finds a divider $receiver.eleCount \geq 1$. Notice that if there does not exist a linear divider, the refiner refines the abstraction of O^+ and O^- by using numerical values from next level in the graph (i.e., *isNull* for *data* and *length* of *data*) and tries again to find a divider. Intuitively, the reason that we look for a divider level-by-level is that we believe that the reason why calling the same method leads to different results is more likely related to the values of variables directly defined in the class and less likely nested in its referenced data variables.

The other scenario where the refiner is consulted is with a candidate query $\mathcal{C}$ and a set of executions which end in the same state in $\mathcal{C}$. Furthermore, extending the executions with a method call corresponding to an event e would result in failure or success. Similar to the case of a membership query, for each execution we obtain a pair $(o, \overrightarrow{p})$ where o is the object state of the main instance prior to the last method call and $\overrightarrow{p}$ is the arguments of the last method call. Similarly, we collect two sets of those pairs O^+ (from those successful executions) and O^- (from those failed executions). Afterwards, SVM is invoked to generate a divider for alphabet refinement.

Algorithm 1 L* Algorithm with Lazy Alphabet Refinement

```
1: Let P = E = {⟨⟩}
2: for e ∈ Σ ∪ {⟨⟩} do
3:     Update T by Q_m(e)
4:     if e needs to be split then
5:         Split(Σ, e, (P, E, T))
6: while true do
7:     while there exists tr·⟨e⟩ where tr ∈ P and e ∈ Σ such
       that tr·⟨e⟩ ≢ tr' for all tr' ∈ P do
8:         P ⟵ P ∪ {tr·⟨e⟩}
9:         for σ ∈ Σ do
10:            tr'' ⟵ tr·⟨e⟩·⟨σ⟩
11:            Update T by Q_m(tr'')
12:            if there is some e' ∈ Σ needs to be split then
13:                Split(Σ, e', (P, E, T))
14:    Construct candidate typestate C from (P, E, T)
15:    if Q_c(C) = 1 then
16:        return C
17:    else
18:        if there is some e' ∈ Σ needs to be split then
19:            Split(Σ, e', (P, E, T))
20:        v ⟵ WS(σ_ce)          ▷ σ_ce is a counterexample
21:        E ⟵ E ∪ {v}
22:        for tr ∈ P and e ∈ Σ do
23:            Update T by Q_m(tr·v) and Q_m(tr·⟨e⟩·v)
24:            if there is some e' ∈ Σ needs to be split then
25:                Split(Σ, e', (P, E, T))
```

C. The Learner

The learner drives the learning process and interacts with both the tester and refiner. It uses an algorithm which extends the L* algorithm [6] with lazy alphabet refinement.

In general, a typestate for a program often requires more expressiveness than DFA and therefore the L* algorithm itself is not sufficient. We solve this problem by extending the L* algorithm with (lazy) alphabet refinement, i.e., by introducing propositions on object states into the alphabet. The details on the extended L* algorithm are presented in the following.

1) L with Lazy Alphabet Refinement:* When the refiner generates a divider ϕ, an event e (which is the event calling some method under certain condition) is effectively divided into two: $[\phi]e$ and $[!\phi]e$. With a modified alphabet, previous learning results are invalidated and therefore learning needs be re-started. However, re-starting from scratch is costly, as we often need multiple rounds of alphabet refinement. In the following, we show how to extend the L* algorithm with lazy alphabet refinement so as to re-use previous learning results as much as possible.

Algorithm 1 shows the pseudo-code of the L* algorithm with lazy alphabet refinement, where $\mathcal{Q}_m(tr)$ denotes the membership query with the trace tr and $\mathcal{Q}_c(\mathcal{C})$ denotes the

Algorithm 2 Split($\Sigma, e, (P, E, T)$)

1: Let ϕ be divider given by the Refiner to refine e
2: $\Sigma \longleftarrow \Sigma \cup \{[\phi]e, [!\phi]e\} \setminus \{e\}$
3: **if** $p \in P$ or $q \in E$ has a substring $\langle e \rangle$ **then**
4: split p into p_1 and p_2 such that p_1 has the substring $[\phi]e$ and p_2 has the substring $[!\phi]e$
5: split q into q_1 and q_2 such that q_1 has the substring $[\phi]e$ and q_2 has the substring $[!\phi]e$
6: Update T by $\mathcal{Q}_m(p_i \cdot q_i)$ for all $i \in \{1, 2\}$
7: **end if**

	$\langle\rangle$
$\langle\rangle$	1
$\langle push \rangle$	1
$^*\langle [!(eleCount \geq 1)]pop \rangle$	0
$^*\langle [eleCount \geq 1]pop \rangle$	1
$^*\langle [!(eleCount \geq 1)]pop, push \rangle$	0
$^*\langle [(eleCount \geq 1)]pop, push \rangle$	0
$^*\langle [!(eleCount \geq 1)]pop, [eleCount \geq 1]pop \rangle$	0
$^*\langle [!(eleCount \geq 1)]pop, [!(eleCount \geq 1)]pop \rangle$	0

Fig. 7. The observation table generated by the lazy L* algorithm.

candidate query of a typestate $\mathcal{C}$. There are two cases where the alphabet refinement takes place: (1) when a membership query triggers the generation of a divider ϕ (lines 5, 13, 25), which means that some alphabet $e \in \Sigma$ needs to be split into $[\phi]e$ and $[!\phi]e$, it calls Algorithm 2 to refine the alphabet and update the corresponding results of the membership queries. (2) A candidate query may also trigger the generation of a divider ϕ (line 19). If so, Algorithm 2 is also called to refine the alphabet and update the corresponding results of the membership queries in the observation table.

We use the Stack example to illustrate the new algorithm. Initially, the alphabet is $\Sigma = \{push, pop\}$. After a series of memberships, Algorithm 1 constructs the first candidate typestate, as shown in Fig. 2 (b), based on the closed and consistent observation table shown in Fig. 2 (a). A candidate query for the first typestate is performed, and the refiner returns a proposition $eleCount \geq 1$ for the positive counterexample $\langle pop \rangle$. The event pop is split into two events: $[eleCount \geq 1]pop$ and $[!(eleCount \geq 1)]pop$, and the L* learning process is restarted from the scratch. Without lazy alphabet refinement, all the membership queries over the new alphabet $\Sigma' = \{push, [eleCount \geq 1]pop, [!(eleCount \geq 1)]pop\}$ have to be queried, as shown in the observation table in Fig. 7. However, with lazy alphabet refinement, only the membership queries marked with a $*$ symbol have to be queried. In this small example, only two membership queries are reduced due to the small alphabet size. In real-world examples, the size of alphabet is usually big, and the number of reduced membership queries is significant. The final typestate learned by Algorithm 1 is the same as the one shown in Fig. 3 (b).

IV. TzuYu Implementation

We have implemented the approach in a tool named TzuYu, which has more than 20K lines of Java code. In this section, we discuss the challenges in implementing the proposed method and how we have addressed them.

We first employ reflection to collect relevant information like fields and methods of each class so as to construct a numerical value graph for each class. The graph of a type depends on the referenced types and hence it may reference many types, but not all referenced types are useful for generating dividers. Therefore we filter classes such as *Thread*, *Exception* and high level interfaces such as *Serializable*. The public methods defined in the target class identify the initial alphabet for the learner. Afterwards, the learner starts to generate membership queries and candidate queries according to Algorithm 1.

Given a membership query, the tester checks whether its abstract trace is feasible or not by generating a number (which is configurable) of executions and uses reflection to run them. During execution, the tester saves the runtime states of the arguments of each method. For argument generation, we develop a just-in-time approach, i.e., generate the required arguments just before executing a method. Some of the chosen arguments may fail the guard condition, and then we choose another argument which can pass the guard condition. If there is no argument satisfying the condition, we generate another set of arguments until the guard condition evaluates to true (or a bound is reached). We don't present the just-in-time algorithm here due to space limitation. Informally, an argument can be obtained from three sources, i.e., randomly generated from a set of pre-defined type compatible values; selected from existing executions that generate type compatible variables; or selected from type compatible out-referenced variables generated by the current execution. The above recursive argument generation procedure may not terminate for a recursive constructor which has a parameter of the same class in which the constructor is defined. We set a maximum call depth for the recursive constructor as did by Lin *et al.* [25].

Before executing each method call, we store the object states of the receiver and the arguments as an instrumented state. We remark that using the Java standard *clone* mechanism to save object states is infeasible because the class may not implement *Serializable* or *Cloneable* interface. We thus implement a *mockup mechanism* similar to the standard clone mechanism in Java to save the runtime object into a mockup object whose tree like class structure resembles the class structure of the original object. The mechanism differs from the standard clone mechanism in that only primitive type values of the object are saved. For reference type field we construct another mockup object as its saved value. These mockup objects can be used by the refiner. When the real object is needed, for instance, to generate a new test, we record the exact sequence of statements whose execution creates the object that can then be used to "clone" the arguments later by re-executing them.

Given a candidate query, the tester generates a number of tests from the typestate. The default number (which is configurable) is twenty multiplied with the maximum length of traces generated in membership queries before this candidate query. Each testing trace is generated by depth first random walking on the typestate up to a fixed length, the length of the trace is set to two plus the maximum length of traces generated during membership queries. Due to randomness in random

TABLE I
THE RUNTIME STATISTICS FOR TZUYU RUNNING THE TARGET CLASSES

Target Class	*LOC*	*#Method*	T_{total}	*#MQ*	*#CQ*	*#Trace*	$\#TC^+$	*#SVM*	T_{SVM}	*#Alphabet*	*#State*
java.util.Stack	50	5	1177	39	4	120	83	4	59	7	2
example.BoundedStack	40	2	764	21	4	98	69	4	138	4	2
java.io.PipedOutputStream	150	5	8343	75	6	200	48	8	5069	9	2
example.PipedOutputStream	40	4	1548	48	5	160	71	5	59	7	2
example.Signature	50	5	3227	75	6	200	102	8	156	9	2

testing and random walking, a test case generated previously may not appear again later. To ensure the learning process is improving always (and hopefully converging), we store all the generated test cases so as to provide consistent answers. Notice that we do not store the instrumented states of the test case to reduce memory consumption and we re-execute the test case to create the states when they are needed (e.g., to evaluate the guard conditions).

One key step in our approach is to automatically generate a divider for alphabet refinement. We use the SVM techniques implemented in LibSVM [8]. The first problem with using SVM is how to choose a good hyperplane as there are in theory an infinite set of hyperplanes which separate two sets of object states. The second problem is that the hyperplane discovered by LibSVM often has float coefficients, which are often not as readable as integer values when we use them to build the typestate. Thus, we always (if possible) choose integer coefficients which constitute a hyperplane which lies between the strongest and weakest hyperplane. Further, we implemented a few heuristics to preprocess the inputs to LibSVM for generating a better divider. Firstly we balance the positive and negative input data sets by duplicating data randomly chosen from the smaller set of the two, as SVM tends to build biased hyperplanes when the input data-set is imbalanced.

Secondly, because the arguments of method calls are generated randomly, LibSVM may generate an incorrect divider. For instance, given a bounded stack with a size bound 5, if *push*(*element*) is invoked with *element* from $\{1, 2, 3\}$ when the bounded stack is full, whereas it is invoked with *element* in $\{5, 6, 7\}$ when the bounded stack is not full. LibSVM may generate a divider *element* ≥ 4 suggesting that calling *push*(*element*) with an input less than 4 will lead to failure. This is obviously incorrect. The problem is avoided with cross validation by checking whether the argument really affects the execution results. This is done by executing the successful (failed, respectively) traces whose arguments are substituted with arguments in the failed (successful, respectively) traces. For instance, in the above example, additional test cases are generated so that every invocation of *push*(*element*) is tested with the same set of input values, i.e., $\{1, 2, 3, 5, 6, 7\}$. As a result, if the argument is irrelevant to the execution result, it will be ruled out by cross validation.

V. EVALUATION

In this section, we first evaluate TzuYu on a set of Java library classes selected from the JDK and then compare TzuYu with existing tools. All the experiments were carried out on a Ubuntu 13.04 PC with 2.67 GHz Intel Core i7 Duo processors and 4 GB memory. All the experimental data is available in our web site [36].

The selected JDK classes (also used in previous related papers [15], [35]) are shown in Table I. Column *LOC* is the size of the class in terms of lines of code. Column *#Method* is the number of methods (excluding the constructors of the target class) which are defined in the target class and used to generate the initial alphabet. In this set of experiments, we generate two values for each parameter in each method. To get a numerical vector from an object state (for SVM consumption), we limit the numerical value graphs to its top five levels, which we found to be sufficient.

A. Results

Table I also shows the statistics of the experiments. Column T_{total} is the total time used in milliseconds. The subsequent three columns show details about the L* algorithms. Column *#MQ* and *#CQ* are the number of membership queries and candidate queries, respectively. Column *#Trace* is the total number of abstract traces generated from random walking. Column $\#TC^+$ is the number of positive concrete test cases generated by TzuYu. Column *#SVM* and T_{SVM} are the total number of SVM calls and the time in milliseconds taken by SVM to generate dividers, respectively. The last two columns show the size of alphabets and the number of states in the final DFA, respectively.

The following observations are made based on the experimental results. Firstly, TzuYu successfully learned typestates in all cases in seconds. Furthermore, in most cases, the time taken by SVM is less than 20% of total time except for *java.io.PipedOutputStream* where the cross validation (in order to determine whether a method parameter is relevant) in a SVM call consumes a few seconds. Secondly, all learned typestates are sound and complete, which we confirm by comparing the learned one with the manually constructed actual one. Thirdly, the number of states in the learned typestate is minimum, i.e., two as we are differentiating two states only: failure or non-failure. This implies that for every method, whether invoking the method leads to failure or not can be determined by looking at the value of the data variables, and further, SVM is able to identify a suitable proposition every time. Lastly, we did not record the memory consumption due to the garbage collection feature of JVM. However, the memory consumption is relatively small since we did not store the instrumented states with the test cases and the number of

TABLE II
PROGRAM INVARIANTS GENERATED BY DAIKON, PSYCO AND TZUYU

Method	Daikon	PSYCO	TzuYu
java.util.Stack.pop()	-	-	$elementCount \geq 1$
java.util.Stack.peek()	-	-	$elementCount \geq 1$
example.BoundedStack.push(Integer)	$size$ one of {0, 1, 2}	-	$size \leq 2$
example.BoundedStack.pop()	$size$ one of {1, 2, 3}	-	$size \geq 1$
java.io.PipedOutputStream.connect(snk)	-	-	$sink == null$ && $snk \neq null$ && $snk.connected == false$
java.io.PipedOutputStream.write(int)	-	-	$sink \neq null$
example.PipedOutputStream.connect(snk)	$sink == null$ && $snk \neq null$ && $snk.connected == false$	$sink == null$ && $snk \neq null$ && $snk.connected == false$	$sink == null$ && $snk \neq null$ && $snk.connected == false$
example.PipedOutputStream.write()	$sink \neq null$	$snk \neq null$	$sink \neq null$
example.Signature.verify()	$Signature.VERIFY == state$	-	$state \geq 2$
example.Signature.sign()	$Signature.SIGN == state$	-	$state \geq 1$&&$state \leq 1$
example.Signature.upate()	$Signature.SIGN \leq state$	-	$state \geq 1$

test cases is relatively small which is linear in the number of candidate queries.

B. Comparison with related tools

We identified three closely related tools. PSYCO [15] is a symbolic execution based typestate learning tool; ADABU [12] is a dynamic behavior model mining framework and Daikon [14] is a dynamic invariant generator. We compare TzuYu with them in terms of time and the quality of the generated models. Table II shows the results of the invariants generated by the three tools and TzuYu. Notice that PSYCO is not available at the time of writing; we thus only obtain the learned typestate documented in their paper [15].

We first compare the learned models as shown in Table II. The invariants generated by ADABU are state invariants and they are omitted from Table II. Methods with the trivial *TRUE* invariant (e.g., *size*() in *Stack*) are also omitted. Both ADABU and Daikon need test cases as input to mine models and therefore we use the test cases generated by TzuYu as their input for a fair comparison. The number of generated test cases for each class is shown in the $\#TC^{+}$ column of Table I. Neither ADABU nor Daikon is able to learn models for all of the classes. For instance, neither mined models for the *java.io.PipedOutputStream* class. ADABU often generates multiple (e.g., dozens of) models for one class, which means ADABU's state abstraction techniques failed to generate a good invariant. The reason is that ADABU employs a set of pre-defined templates to generate invariants. If a mined state invariant contains irrelevant variables, ADABU's state abstraction and model merging technique fails and therefore no unified model is generated. Daikon failed to mine models for *java.util.Stack* class. Both ADABU and Daikon use pre-defined invariant templates. In comparison, the typestates (which are invariants) generated by TzuYu are better because TzuYu does not rely on templates but rather uses SVM techniques to discover propositions dynamically based on the object states. Furthermore, Daikon uses only successful executions whereas TzuYu uses both successful and failed executions, thus the model learned by TzuYu is more accurate than the one generated by Daikon.

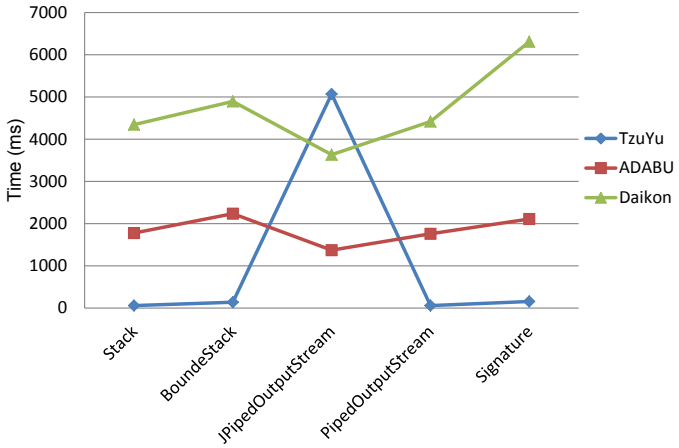


Fig. 8. Time consumed in milliseconds to mine models for target classes.

For *example.PipedOutputStream* and *example.Signature*, PSYCO [15] can learn accurate transition guards due to the fact that it encodes all path conditions in the source code and uses an SMT solver to exactly find out whether failure happens. However PSYCO is limited by the capability of the SMT solver.

Next, we compare the execution time of each tool on mining the models briefly. The time taken by each tool to mine the models is plotted in Fig. 8. PSYCO is not available for running the target classes, we cannot get the time for it. Both ADABU and Daikon need test cases while TzuYu generates the test cases, so we only include the time consumed by SVM for TzuYu. The figure shows that TzuYu often uses less time in generating the models. An exception is the *java.io.PipedOutputStream* class for the reason mentioned above.

C. Limitations of TzuYu

Firstly, because our approach is based on testing, there is no guarantee that the learned typestate is sound or complete. However, this can be fixed to certain extent by using an SMT solver to verify the learned typestate. For instance, the typestate for Stack in Fig. 3 (b) can be verified by showing that each transition is sound and complete, e.g., the self-looping transition at state 1 labeled with $[eleCount \geq 1]pop$ can be verified

by proving two Hoare triples: $\{eleCount \geq 1\}pop()\{noerror\}$ (executing *pop* with a pre-condition $eleCount \geq 1$ will not lead to error) and $\{eleCount \leq 0\}pop()\{error\}$. Further, if the SMT solver identifies a counterexample, the counterexample can be used to refine the typestate.

Secondly, because our approach is based on random testing, there is no guarantee that a good divider can be discovered in general—though it should emerge in theory after sufficient testing. This can be partially fixed if we can obtain "better" test cases through different means, e.g., from real execution history of the given class, or through more sophisticated test case generation methods like concolic testing [32] and combinational testing [20].

Thirdly, our method will not terminate if the typestate for the class under analysis is beyond the expressiveness of finite-state machines with linear guard conditions. If the refiner fails to find a divider for a membership query with conflicting results (i.e., the same sequence of events leads to failure and success), a counterexample (i.e., a path which is predicated to fail by the typestate but succeeds in real testing execution, or the other way round) is returned so that L* may introduce a new state. In the worst case, TzuYu will keep generating typestates with ever growing number of states (and eventually times out). This is due to the limitation of SVM that could be overcome using advanced learning techniques.

VI. Related Work

Our approach is related to specification mining. We refer interested readers to the book by Lo *et al.* [26] for a comprehensive literature review. Therefore, we only review previous work that is closely related to the three components in TzuYu and the overall approach.

The idea of using testing as the teacher for L* algorithm is also found in the AMC approach [16] which uses L* to handle counterexamples returned by the model checker. The L* algorithm is also used for learning assumptions in compositional verification [3], [7], [19], [24] by formal methods community. TzuYu differs from these work in that it uses L* algorithm to learn the specification from source code.

The idea of learning interface specifications from source code was proposed by Alur *et al.* [4] which learns interface specifications from source code automatically by using a model checker as the teacher. The PSYCO tool [15] achieves the same goal by using a symbolic execution engine as the teacher. The X-PSYCO [18] tool extends PSYCO by answering membership and candidate queries with testing under inputs generated from symbolic execution. In comparison, TzuYu employs testing and thus avoids expensive model checking or symbolic execution. Similarly, Aarts *et al.* [1] proposed a fully automated data abstraction technique to learn a restricted form of Mealy machine in which only testing equality of arguments is allowed. TzuYu's SVM based alphabet refinement can be applied to more programs.

Our testing strategy is related to Randoop [28]. We extend Randoop to the context of learning in which the receiver object must be the same in order to learn a better model and we also add a new source for reference arguments which can be chosen from an out-reference variables to improve data coverage. Tester in TzuYu is also related to TAUTOKO [11] which generates more test cases by mutating existing traces in the mined model (by using ADABU) to augment the model learning process as well as finding bugs.

We extend the active learning L* algorithm with lazy alphabet refinement. There are also other learning algorithms such as sk-strings algorithm [30]. The sk-strings algorithm passively learns a DFA from a given set of traces by generalizing the method call sequences in the trace to form the final DFA. ADABU [12] can be classified as a passive learner which requires a set of test cases as input; it abstracts the concrete states with simple templates to abstract states thus to get the abstract traces and then it merges models from abstract traces to generate a model. The combination of an active learning algorithm with automatic argument generation techniques enables TzuYu to learn stateful typestates automatically.

The refiner in TzuYu is inspired by Sharma *et al.* [33] who use SVM and SMT solver to generate interpolants for counterexamples produced by model checkers. The goal of the refiner is in line with that of the dynamic invariant generator Daikon [14] and Axiom Meister [35]. Daikon uses a set of pre-defined invariant templates over data from the set of given runtime traces. Daikon may find some irrelevant invariants at a program point. Axiom Meister uses symbolic execution to collect all the path conditions which are then abstracted into preconditions. TzuYu's refiner is based on SVM which enables TzuYu to find relevant linear arithmetic propositions over a large number of variables.

VII. Conclusion and Future Work

Despite the recent progress on learning specifications from various software artifacts, the community is still challenged with difficulties in dealing with data abstraction for common programs. In this paper, we propose a fully automated typestate learning approach from source code. To fully automate the generation of test cases which are the required inputs for many automata learning tools, we combine the active learning algorithm L* with a random argument generation technique. We then use a supervised machine learning algorithm (i.e., the SVM algorithm) to abstract data into propositions.

For the future work, we want to use symbolic execution to ensure that the learned model is sound and try other machine learning techniques in order to generate better dividers. We also want to evaluate the effectiveness of different test case generation techniques in learning setting.

Acknowledgements We thank the anonymous reviewers for their invaluable comments. This work is partially supported by NTU-NAP project "Formal Verification on Cloud" and TRF project "Research and Development in the Formal Verification of System Design and Implementation". This work is also supported by project "IDD11100102A/IDG31100105A" from Singapore University of Technology and Design.

REFERENCES

[1] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. Vaandrager. Automata learning through counterexample guided abstraction refinement. In *FM*, pages 10–27, 2012.

[2] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE*, pages 25–34, 2007.

[3] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562. Springer Berlin Heidelberg, 2005.

[4] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109, 2005.

[5] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

[6] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987.

[7] H. Barringer and D. Giannakopoulou. Proof rules for automated compositional verification through learning. In *In Proc. SAVCBS Workshop*, pages 14–21, 2003.

[8] C.-C. Chang and C.-J. Lin. LIBSVM: a library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, 2011.

[9] T. Chow. Testing software design modeled by finite-state machines. *Software Engineering, IEEE Transactions on*, SE-4(3):178–187, 1978.

[10] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279, 2000.

[11] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA*, pages 85–96, 2010.

[12] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA*, pages 17–24, 2006.

[13] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Trans. Softw. Eng.*, 31(12):1056–1073, Dec. 2005.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. Mccamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, 2006.

[15] D. Giannakopoulou, Z. Rakamaric, and V. Raman. Symbolic learning of component interfaces. In *SAS*, pages 248–264, 2012.

[16] A. Groce, D. Peled, and M. Yannakakis. AMC: an adaptive model checker. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 521–525. Springer, 2002.

[17] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[18] F. Howar, D. Giannakopoulou, and Z. Rakamaric. Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In *ISSTA*, pages 268–279, 2013.

[19] K. Ji, Y. Liu, S.-W. Lin, J. Sun, J. S. Dong, and T. K. Nguyen. CELL: A compositional verification framework. In *ATVA*, 2013. To appear.

[20] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94–96, 2009.

[21] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *Proc. of the IEEE*, 84(8):1090–1123, 1996.

[22] S.-W. Lin, É. André, J. S. Dong, J. Sun, and Y. Liu. An efficient algorithm for learning event-recording automata. In *ATVA*, pages 463–472, 2011.

[23] S.-W. Lin and P.-A. Hsiung. Counterexample-guided assume-guarantee synthesis through learning. *IEEE Transactions on Computers*, 60(5):734–750, 2011.

[24] S.-W. Lin, Y. Liu, J. Sun, J. Dong, and É. André. Automatic compositional verification of timed systems. In *FM*, pages 272–276. 2012.

[25] Y. Lin, X. Tang, Y. Chen, and J. Zhao. A divergence-oriented approach to adaptive random testing of java programs. In *ASE*, pages 221–232, 2009.

[26] D. Lo, K. Cheng, and J. Han. *Mining software specifications: methodologies and applications*. Chapman and Hall/CRC Data Mining and Knowledge Discovery Series. Taylor & Francis Group, 2011.

[27] M. G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *OOPSLA*, pages 77–96, 2005.

[28] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA*, pages 815–816, 2007.

[29] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE*, pages 371–382, 2009.

[30] A. Raman and J. Patrick. The sk-strings method for inferring PFSA. In *ICML*, 1997.

[31] B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors. *Advances in kernel methods: support vector learning*. MIT Press, 1999.

[32] K. Sen, D. Marinov, and G. Agha. CUTE: a Concolic Unit Testing Engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[33] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV*, pages 71–87, 2012.

[34] R. E. Strom and S. Yemini. Typestate: a programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[35] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. In *ICFEM*, pages 717–736, 2006.

[36] H. Xiao. TzuYu hosting site. http://bitbucket.org/spencerxiao/tzuyu, May 2013.

[37] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, pages 307–318, 2009.