TLV: Abstraction through Testing, Learning and Validation

Jun Sun[1], Hao Xiao[2], Yang Liu[2], Shang-Wei Lin[1], and Shengchao Qin[3]
[1]ISTD Pillar, Singapore University of Technology and Design, Singapore
[2]School of Computer Engineering, Nanyang Technological University, Singapore
[3]School of Computing, Teesside University, United Kingdom

ABSTRACT

A (Java) class provides a service to its clients (i.e., programs which use the class). The service must satisfy certain specifications. Different specifications might be expected at different levels of abstraction depending on the client's objective. In order to effectively contrast the class against its specifications, whether manually or automatically, one essential step is to automatically construct an abstraction of the given class at a proper level of abstraction. The abstraction should be correct (i.e., over-approximating) and accurate (i.e., with few spurious traces).

We present an automatic approach, which combines testing, learning, and validation, to constructing an abstraction. Our approach is designed such that a large part of the abstraction is generated based on testing and learning so as to minimize the use of heavy-weight techniques like symbolic execution. The abstraction is generated through a process of abstraction/refinement, with no user input, and converges to a specific level of abstraction depending on the usage context. The generated abstraction is guaranteed to be correct and accurate. We have implemented the proposed approach in a toolkit named TLV and evaluated TLV with a number of benchmark programs as well as three real-world ones. The results show that TLV generates abstraction for program analysis and verification more efficiently.

1. INTRODUCTION

Abstraction is perhaps the single most powerful weapon for combating the complexity in program analysis and verification. A good abstraction of a program should be at a proper level of abstraction, which is decided by the usage context. It should have a much smaller state space so that it is subject to efficient search-based analysis like model checking [15]. It should be an over-approximation of all behaviors of the program so that we could conclude that the given program satisfies a (safety) property if the abstraction does. It should be sufficiently accurate so that analysis based on the abstraction would result in few false alarms. *The challenge is: how do we automatically construct such an abstraction?*

In this work, we propose an automatic approach called TLV which combines testing, learning, and validation to generating an abstraction of a given Java class. The abstraction characterizes behaviors of any object of the class. In a way, TLV is designed to mimic programmers so as to combat the complexity of program analysis and verification. When experienced programmers are asked to analyze a given program, they often execute the program with various inputs, from which (among other artifacts like documentations, program comments, and domain knowledge) they would form some initial idea on what the program does (and how) and then validate their guess with more test cases or through code review. They may guess a number of times until they build a correct abstraction (in the mind) on what the program does. Depending on their objective, they would stop the process once the abstraction allows them to accomplish their analysis goal.

The workflow of TLV is inspired by the above process, as shown in Figure 1. The inputs are the source code of a program and optionally an artifact which TLV could use to determine the proper level of abstraction. TLV has three phases: learning, validation and refinement. In the learning phase, we apply automatic testing techniques to generate, inexpensively, sample behavior of the class, which consists of sequences of method calls. The hope is that the test cases would cover a large portion of the complete behavior. Furthermore, we adopt techniques from the machine learning community and design a learning algorithm based on the L^* algorithm [2] to not only guide the test case generation but also generate candidate abstractions systematically based on the testing results. In the validation phase, we apply more heavy-weight techniques like symbolic execution to validate the abstraction so that the abstraction is guaranteed to be correct and accurate. After validation, the abstraction is checked to see whether it is at a proper level of abstraction. If it is too abstract, we refine the abstraction and restart from the testing phase. The iterative process ends when the correct and accurate abstraction is constructed.

However, a correct abstraction could be completely trivial and thus useless. In order to make sure the abstraction is useful, we need to answer two questions. The first question is: what is the right model for the abstraction? The answer decides what kind of behaviors the abstraction is capable of capturing, which in turn defines what purposes the abstraction could serve. One form of program abstraction is predicate abstraction [5] which is particularly useful for analyzing programs with non-trivial data states . Given a program and a set of predicates, predicate abstraction constructs an abstraction of the program by focusing only on the truth values of a set of predicates. In our setting, predicate abstraction means to construct an abstraction of the class in the form of a labeled Kripke structure [11], i.e., a finite state automaton whose transitions are labeled with method names and whose states are labeled with predicates. An example is shown in Figure 5(b). Compared with other models like finite state automata, this model is more expressive (for instance, using a predicate on the number of elements in a stack, it can express languages like the number of pop operations must be less than or equal to the number of $push$ operations) and more catered for classes with rich data states. Furthermore, such models can be readily fed into a model checker for verification.

The second question is what level of abstraction is sufficient for the analysis. Equivalently, in the context of predicate abstraction, what is the set of predicates? This question can be answered only based on the usage context. TLV provides three different ways. First, if the abstraction is used to verify whether the class satisfies certain temporal logic formula, it must be at a level which would allow us to either prove or disprove the property. TLV extracts from the formula an initial set of predicates and then generates an abstraction as accurate as possible with respect to the predicates. Afterwards, the abstraction can be verified against the given property. In the event that a spurious counterexample is found, TLV provides a way of automatically identifying a candidate predicate for abstraction refinement, based on the testing results and machine learning techniques. With the new predicate, a new abstraction is constructed and this process repeats until TLV generates an abstraction which either proves or disproves the property. However if the abstraction is used by humans, the users should be able to customize the level of abstraction and TLV provides two ways to set the abstraction level manually. That is, users can either provide a set of predicates; or users can choose to provide no predicate initially, but then ask TLV to resolve non-determinism in the abstraction through automatically generating new predicates.

The underlying idea of TLV is simple. The task of abstracting a class is to discover all of its behavior, whereas testing could be effective in uncovering behavior. Therefore, we first apply testing techniques with the hope to discover a large part of the behavior inexpensively. However, simply relying on random testing is limited (e.g., for predicate coverage [47]) and thus active learning techniques are adopted to not only guide the testing process but also to construct concise candidate abstractions automatically. Only when a likely abstraction has been obtained, theorem proving techniques are used to validate the abstraction. Furthermore, through learning, we are able to automatically discover predicates which can be used to refine the abstraction. The idea of learning from traces of a program is not new [9, 22, 28, 49]. Neither is the idea of verifying the learned model against programs [3, 50]. Rather, TLV combines a number of techniques for effective abstraction.

In short, we make the following technical contributions. First, we develop an approach on combining testing, active learning, and validation to construct predicate abstractions at the proper level of abstraction. Second, we propose a way of generating new predicates to refine a predicate abstraction. Third, we integrate our approach in a toolkit called TLV. We evaluate our approach using a number of programs, including benchmark programs as well as three real-world classes, and show that TLV generates abstraction efficiently for program analysis and verification.

2. AN ILLUSTRATIVE EXAMPLE

In this section, we illustrate how TLV works using a simple example. The only input to TLV is the bounded stack class shown in Figure 2. For simplicity, we focus on two methods: $push$ and pop. Recall that we need a usage context in order to determine the right level of abstraction. For now, assume that the abstraction is to be used for human comprehension and the user chooses not to provide any predicate initially. Based on the assumption above, the initial set of predicates is $\{\top, \bot\}$ where $\bot$ is a special default predicate which denotes whether a failure (i.e., assertion violation or un-handled exception) has occurred and $\top$ denotes no failure.

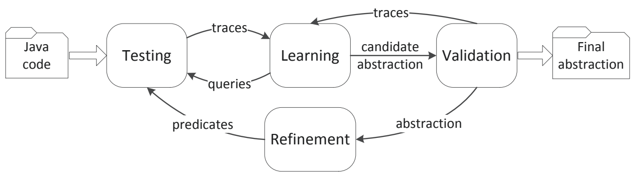


Figure 1: The high-level workflow of the TLV

```
public class BoundedStack {
  private static final int MAX_SIZE = 1024;
  private int size;
  private int[] elements;

  public BoundedStack() {
    size = 0;
    elements = new int[MAX_SIZE];
  }
  public void push(int element) {
    if (size >= MAX_SIZE) {
      throw new IllegalStateException("Full Stack");
    }
    elements[size] = element;
    size++;
  }
  public int pop() {
    if (size <= 0) {
      throw new IllegalStateException("Empty Stack");
    }
    size--;
    return elements[size];
  }
}
```

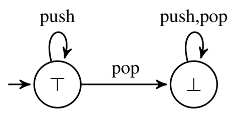
Figure 2: A bounded stack in Java

The Learning Phase In the learning phase, TLV applies a learning algorithm similar to the L* algorithm [2] to learn a candidate abstraction, relying on automatic testing techniques [37]. TLV drives the learning process by generating two kinds of queries (both of which are slightly different from those in the L* algorithm). One is membership queries, i.e., whether a sequence of method calls would result in a particular abstract state. The other is candidate queries, i.e., whether a candidate abstraction is correct and accurate (formally defined in Section 3). The queries and testing results are summarized in an observation table, as shown in Figure 3 (a) where $\langle\rangle$ is an empty sequence of method calls; $\langle pop, push\rangle$ denotes the sequence of calling $push$ after pop. The result column shows the abstract state after the corresponding method calls. For instance, after an empty sequence of method calls, $\top$ is true and calling pop right after initialization results in exception, i.e., $\bot$. Notice that because methods may take parameters, the same sequence of method calls may result in different abstract states, as we shall see later. Based on the observation table, TLV generates the first candidate abstraction, as presented in Figure 3 (b).

Next, TLV asks a candidate query: is the abstraction in Figure 3 (b) correct? To answer the query, TLV performs random walking, i.e., randomly generates a set of tests which correspond to traces of the abstraction. Through the random walking, one inconsistency between the abstraction and the class under analysis is identified. That is, the abstraction predicts that calling pop from state $\top$ always results in $\bot$, whereas it is not the case. For instance, calling method $push$ first and then pop results in no failure. The inconsistency suggests that the abstraction must be modified. In this case, the observation table is updated, as shown in Figure 3 (c), which includes the sequence $\langle push, pop\rangle$ and its testing result. After more membership queries, TLV constructs the candi-

trace	result
$\langle\rangle$	$\top$
$\langle pop\rangle$	$\bot$
$\langle push\rangle$	$\top$
$\langle pop, push\rangle$	$\bot$
$\langle pop, pop\rangle$	$\bot$

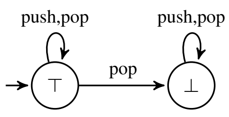
(a)



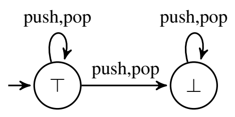
(b)

trace	result
$\langle\rangle$	$\top$
$\langle pop\rangle$	$\bot$
$\langle push\rangle$	$\top$
$\langle push, pop\rangle$	$\top$
$\langle pop, push\rangle$	$\bot$
$\langle pop, pop\rangle$	$\bot$

(c)



(d)



(e)

Figure 3: Artifacts in the 1st learning and validation iteration

date abstraction shown in Figure 3 (d). The answer to the candidate query is positive and thus the learning phase terminates.

The Validation Phase The candidate abstraction may not be correct due to limitations of random testing. For instance, the abstraction in Figure 3 (d) is not correct as invoking method $push$ at state $\top$ may result in $\bot$ when the size of the stack equals MAX_SIZE. This behavior is missing because there is no test case which invokes $push$ more than 1024 times. In general, cases like this are hard to generate through random testing. Thus, the learned abstraction must be validated and refined if necessary. For the candidate abstraction shown in Figure 3 (d), two proof obligations are generated. One is $\{\top\}push\{\top\}$ (a Hoare triple), which denotes that invoking $push$ when there is no failure always results in no failure. The other is $\{\top\}pop\{\top \lor \bot\}$, i.e., invoking pop at $\top$ may or may not result in failure. We adopt the assertion checking feature in Symbolic PathFinder (SPF) [41] to discharge proof obligations. First, TLV modifies the $push$ method by enclosing its method body with a try block; it adds $assert(false)$ to the catch block, i.e., to assert that there is no failure and adds $assert(true)$ to the finally-block, i.e., to assert the post-condition; it adds the pre-condition to an if-conditional. The modified $push$ method is shown in Figure 4. Then TLV symbolically executes the modified $push$ with both parameters $element$ and $size$ as symbolic inputs. An assertion violation is found with concrete values for the symbolic inputs: $element = 3$ and $size = 1024$. Using the concrete values as parameters for $push$, TLV constructs a test case and executes $push$, which results in an exception (i.e., $\bot$). Thus a transition from state $\top$ to $\bot$ is added to the abstraction. After the proof obligation $\{\top\}pop\{\top\lor\bot\}$ is also discharged, the abstraction shown in Figure 3 (e) is guaranteed to be correct and accurate (see the proof in Section 3).

Learning a candidate abstraction helps to reduce the proving effort. If a naïve approach was used to abstract the class, we would need to check satisfiability of every combination $\phi \land m \land \delta$, i.e., whether invoking m with ϕ results in a state satisfying δ, where ϕ and δ are constraints which can be formed using conjunction of the given predicates or their negations and m is a method. The number of such combinations is exponential to the number of predicates.

The Refinement Phase A nondeterministic abstraction like Figure 3(e) might be confusing if it is intended for humans. For instance, what does it mean to say that calling pop may or may not lead to failure? To resolve non-determinism like this, TLV can be instructed to identify predicates which would explain, for instance, when exactly calling pop leads to failure. The standard approach (e.g., as in [14]) is to partition the state $\top$ based on $wp(pop, \bot)$, i.e., the weakest precondition of pop resulting in exception. Computing weakest precondition is often expensive. Instead, TLV applies machine learning techniques, e.g., Supporting Vector Machines (SVMs) [42], to identify a new predicate. In particular, TLV gathers two groups of object states based on the test cases at state $\top$. One group contains stack objects which would result in state $\bot$ after invoking pop. The other group contains those which would result in $\top$. TLV uses SVM to generate a predicate which partitions the two groups. The generated predicate is: $2 * size \geq 1$, which is turned into $size > 0$ after some bookkeeping (based on the fact that $size$ is an integer). With the new predicate, we repeat the learning and validation phase and obtain the abstraction in Figure 5(b).

```
public void push(int element, int size) {
  if (true) {//true encodes the pre-condition
    try {
      if (size >= MAX_SIZE) {
        throw new Exception("Full Stack");
      }
      elements[size] = element;
      size++;
    } catch (Exception e) { assert(false);
    } finally { assert(true); }//post-condition
  }
}
```

Figure 4: Modified *push* method

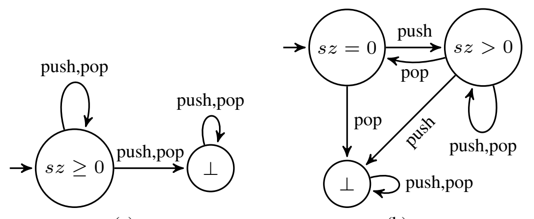


Figure 5: Refined abstraction, where sz stands for the field $size$

The level of abstraction can be determined for different usage contexts. For instance, if a temporal logic property is present (i.e., to be verified), TLV would generate and refine the abstraction based on interactions with the model checker. For instance, assume the property is $\mathbf{G}(push \land \mathbf{X}pop \implies \mathbf{X}(size \geq 0))$ (written in state/event linear temporal logic [11]), i.e., after $push$ and pop, $size \geq 0$ should be always true. The initial set of predicates is set to be $\{size \geq 0, \bot\}$, i.e., all predicates in the property plus the default one $\bot$. Through the learning and validation phase, we obtain the abstraction shown in Figure 5(a). Through model checking (taking the abstraction as a labeled Kripke structure [11]), we found a spurious counterexample: $\langle size \geq 0, push, size \geq 0, pop, \bot\rangle$, which is a run of the abstraction. To remove this spurious counterexample, again the standard approach is to partition the state p based on $wp(pop, \bot)$. TLV rather applies SVM to identify a new predicate for differentiating states from which invoking pop results in state $\bot$ from states resulting in $size \geq 0$. With the generated predicate $size > 0$, TLV generates a new abstraction shown in Figure 5(b). We remark that the spurious counterexample above is ruled out by the new abstraction. Model checking the abstraction against $\mathbf{G}(push \land \mathbf{X}pop \implies \mathbf{X}(size \geq 0))$ is successful and thus this abstraction serves as a proof of the property at an abstraction level which is more abstract than the code.

3. THE TLV APPROACH

In this section, we present the details on how TLV generates an abstraction. We start with defining the problem.

3.1 Problem Definition

We assume a Java class $\mathcal{C}$ contains a finite set of instance variables V and a finite set of methods M, each of which may update variables in V. The semantics of $\mathcal{C}$ is a labeled transition system (S_c, s_c, M, T_c) where S_c is a set of states, each of which is a valuation of all variables in V; $s_c \in S_c$ is the initial state; $T_c : S_c \times M \times S_c$ is the transition relation such that $(s, m, s') \in T_c$ iff, given the variable valuation s, executing method m may result in variable valuation s'. A run of the labeled transition system (a.k.a. a test of $\mathcal{C}$) is a finite sequence of alternating states and transitions $\langle s_0, m_0, s_1, m_1, \cdots, m_k, s_{k+1}\rangle$ such that $s_0 = s_c$ and $(s_i, m_i, s_{i+1}) \in T_c$ for all $i \geq 0$. The sequence of method calls in the run $\langle m_0, m_1, \cdots, m_k\rangle$ is called a trace.

The problem is to construct an abstraction of $\mathcal{C}$ automatically. Let $Prop$ be a set of propositions constituted by variables in V. We write 2^{Prop} to denote the set of predicates each of which is the conjunction of a subset of propositions in $Prop$ and the negation of the rest. For instance, if $Prop = \{p, q\}$, 2^{Prop} is $\{p \wedge q, p \wedge \bar{q}, \bar{p} \wedge q, \bar{p} \wedge \bar{q}\}$. We write the powerset of 2^{Prop} as $\wp 2^{Prop}$, i.e., the set of all subsets of 2^{Prop}. A member of $\wp 2^{Prop}$ can be represented succinctly. For instance, the set $\{p \wedge \bar{q}, p \wedge q\}$ can be represented as $\{p\}$, i.e., their disjunction. We write $s \models \phi$ to denote that ϕ evaluates to true given the variable valuation s. Given a set of concrete states X, we write $abs_{Prop}(X)$ to denote the disjunction of all members ϕ of 2^{Prop} such that $s \models \phi$ for some $s \in X$. For instance, if $Prop = \{size \geq 0, size \geq 1024\}$, $abs_{Prop}(\{size \mapsto 5, size \mapsto 1034\})$ is $size \geq 0$.

An abstraction of $\mathcal{C}$ w.r.t. $Prop$, denoted as $\mathcal{A}$, is a labeled transition system (S_a, s_a, M, T_a) where $S_a \subseteq \wp 2^{Prop} \cup \{\bot\}$ is a set of abstract states, each of which is a subset of 2^{Prop} or $\{\bot\}$ (a special state denoting exception); $s_a \in S_a$ satisfies $s_c \models s_a$; $T_a \subseteq S_a \times M \times S_a$ is an abstract transition relation. The abstraction is *correct* if there exists $(s, m, s') \in T_c$ such that $s \models \phi$ and $s' \models \phi'$ imply $(\phi, m, \phi') \in T_a$. The abstraction is *accurate* if for all $(\phi, m, \phi') \in T_a$ implies there exists $(s, m, s') \in T_c$ such that $s \models \phi$ and $s' \models \phi'$. However, a correct and accurate abstraction may still contain spurious runs, due to broken traces [25] (i.e., an abstract transition might be feasible locally but not globally). We use abstract states and predicates interchangeably hereafter.

A naïve approach to obtaining $\mathcal{A}$ is to check whether every possible transition (ϕ, m, ϕ') where $\{\phi, \phi'\} \subseteq S_a$ and $m \in M$ is contained in $\mathcal{A}$. This is infeasible as in the worst case there are $2^{|Prop|} \times |M| \times (2^{|Prop|} + 1)$ checks, where $|Prop|$ is the number of propositions and $|M|$ is the number of methods. Thus, we propose the process shown in Figure 1 to learn $\mathcal{A}$.

3.2 Testing and Learning

TLV starts with a testing and learning phase to obtain a candidate abstraction inexpensively. In this phase, TLV can be viewed as a 'game' between two players. One is a learner who, in order to learn, asks a series of membership queries and candidate queries. A member query asks which abstract states can be reached after a trace. For instance, in the stack example, a membership query would be: $\langle push, pop\rangle$. After multiple membership queries, the learner makes a guess on what the abstraction is by *generalizing* what it has learned so far and asks a candidate query. A candidate query asks whether a candidate abstraction is correct and accurate. The other player is a teacher. The teacher's job is to answer both kinds of queries. Ideally, a teacher would answer a membership query with all abstract states that can be reached with the given trace. The teacher answers positively to a candidate query iff the candidate abstraction is correct and accurate; if the answer to a candidate query is negative, the teacher should provide a counterexample in the form of a concrete test case, which shows the candidate abstraction is problematic. In practice, having a perfect teacher is expensive. For instance, answering a membership query would require checking whether it is feasible to satisfy any proposition in $Prop$ after a sequence of method calls, which is a non-trivial reachability analysis problem. Even worse, answering a candidate query would require solving the abstraction problem itself. Thus, instead of using a perfect teacher, TLV employs a tester (i.e., an imperfect teacher) to answer the queries.

Algorithm 1: The learning algorithm

```
Input: a program and a set of propositions $Prop$
Output: an abstraction
1  let obs be an empty observation table; $visited$ be $\emptyset$;
2  while true do
3      while obs is not closed and the time is not up do
4          let trace $tr$ s.t. $T(tr) \neq T(tr')$ $\forall$ prefix $tr'$ of $tr$;
5          for $m \in M$ do
6              generate a membership query $tr \cdot \langle m\rangle$;
7              let $X := Randoop(tr \cdot \langle m\rangle)$;
8              $obs := obs + (tr \cdot \langle m\rangle \mapsto abs_{Prop}(X))$;
9      generate a candidate query $\mathcal{A}$ from obs;
10     apply random walking to check $\mathcal{A}$;
11     if no inconsistency found then
12         if Algorithm 2($\mathcal{A}$, obs, visited) returns true then
13             return $\mathcal{A}$;
14     else
15         let $(tr, s)$ be a counterexample to the candidate $\mathcal{A}$;
16         $obs := obs + (tr \mapsto abs_{Prop}(\{s\}))$;
```

TLV's algorithm is presented as Algorithm 1. The inputs are a program and a set of propositions $Prop$ and the output is an abstraction. TLV maintains two data structures. One is an *observation table obs* for storing (abstract) testing results and the other is a set $visited$ for storing validation results. The observation table *obs* is a tuple (P, E, T) where $P \subseteq M^*$ is a set of traces; $E \subseteq S_a$ is a set of abstract states; $T : P \rightarrow E$ is a mapping function such that $T(tr) = \phi$ means that after the trace tr, the abstract state ϕ can be reached. Initially, P, E, T, and $visited$ are all empty (line 1). We write $obs := obs + (tr \mapsto \phi)$ to denote the operation of adding the mapping $tr \mapsto \phi$ into the table, i.e., replacing P with $P \cup \{tr\}$; replacing E with $E \cup \{\phi\}$; T is updated with $T(tr) := \phi$ if tr was not in the domain of T; otherwise, $T(tr) := T(tr) \vee \phi$. Intuitively, the latter states that if we knew that after tr, we can reach an abstract state $T(tr)$, with the new mapping $tr \mapsto \phi$, we now know that after tr, we can reach either $T(tr)$ or ϕ.

Within a certain time limit, TLV tries to make the observation table *closed* by asking multiple membership queries and adding mappings into *obs* (line 3–8). Note that the concept of *consistency* in the *L** algorithm is irrelevant in our setting. An observation table is *closed* if the set P is prefix-closed and for all $tr \in P$ such that tr is not a prefix of some other trace in P (i.e., tr is maximum), there always exists a prefix of tr say $tr' \in P$ such that $T(tr') = T(tr)$. Intuitively, the latter means that tr can be represented by its prefix and therefore TLV does not need to test further. Since there are only finitely many abstract states, tr would eventually visit a state which is visited by its prefix. We remark that this definition is justified because our goal is to discover as many abstract states and transitions as possible. If the observation table is not closed, there must be a trace tr such that $T(tr)$ is not equivalent to $T(tr')$ for every prefix tr' of tr. In such a case, a membership query (i.e., $tr \cdot \langle m \rangle$) is generated for each method (line 6). In order to answer the query inexpensively, TLV generates multiple test cases using random testing (line 7). Function $Randoop(tr)$ is similar to the Randoop algorithm [37]. Given a membership query tr, TLV generates multiple test cases calling the methods in the query one-by-one (from the initial concrete state). In general, the methods would have multiple parameters and TLV generates arguments for every method call. Given a typed parameter, TLV randomly generates a value from a pool of type-compatible values. This pool composes of a set of predefined values (e.g., a random integer for an integer type, $null$ or an object with the default object state for a user-defined class) and type-compatible objects that have been generated during the testing process. In order to re-create the same object, we store the test case which produces the object.

After generating and executing multiple test cases according to $tr \cdot \langle m \rangle$, TLV collects the concrete data states reached by the test cases (say X) and updates the observation table with the mapping $T(tr \cdot \langle m \rangle) = abs_{Prop}(X)$ (line 8). Ideally, after multiple membership queries, once the observation table (P, E, T) is closed, TLV constructs a candidate abstraction $\mathcal{A} = (S_a, s_a, M, T_a)$ such that $S_a = E$; s_a is the state corresponding to the empty trace $T(\langle\rangle)$; $(\phi, m, \phi') \in T_a$ if there exists $tr \in P$ and $m \in M$ such that $T(tr) = \phi$ and $T(tr \cdot \langle m \rangle) = \phi'$. In practice, with many methods in the class, it might take a long time before the observation table is closed. Nonetheless, with the validation phase, we can construct the candidate abstraction even if the observation table is not closed. In fact, the goal is to discover every abstract behavior of the class and it is guaranteed that every behavior is discovered either during testing or validation. Thus, if closing the observation table takes a long time, TLV times out and constructs $\mathcal{A}$ based on *obs*.

Once the observation table is closed or learning timeouts, TLV raises a candidate query on whether $\mathcal{A}$ is correct and accurate w.r.t. $Prop$ (line 9). TLV then employs a slightly different testing technique to answer candidate queries. We associate each abstract state ϕ in $\mathcal{A}$ with a set of concrete states which have been generated through testing so far and satisfy ϕ. Based on these concrete states, TLV uses random walking to construct test cases from *each abstract state* in $\mathcal{A}$ to further explore behaviors of $\mathcal{C}$ (line 10). The testing result is then compared with $\mathcal{A}$ to see whether they are consistent. $\mathcal{A}$ is consistent with the testing result iff for any sequence of method calls tr' from a concrete state (associated with an abstract state ϕ), the resultant concrete states X are consistent with the corresponding abstract state ϕ' reached by the same sequence of methods in $\mathcal{A}$, i.e., $abs_{Prop}(X)$ logically implies ϕ'. There is an inconsistency iff there exists a concrete state $s \in X$ such that $s \not\models \phi'$ (line 11). In such a case, TLV constructs a pair (tr, s), where $tr = tr_1 \cdot tr'$ and tr_1 is the shortest trace reaching ϕ in $\mathcal{A}$, as a counterexample to the candidate query (line 15), which is then used to update the observation table (line 16). For instance, assume $Prop = \{size \geq 0\}$ and the abstract state after tr in the observation table is $size \geq 0$, i.e., $T(tr) = size \geq 0$. If after calling the methods in tr in sequence, the concrete states are $\{size \mapsto 2, size \mapsto 3, size \mapsto 4\}$, then it is consistent. However, a testing result $size \mapsto -2$ would be an inconsistency and the observation table would be updated so that $T(tr) = size \geq 0 \vee size < 0$.

Once the observation table is updated, TLV again checks whether it is closed and raises membership queries if it is not, until the next candidate query is generated. Once the tester answers positively to a candidate abstraction (at line 11), TLV obtains an abstraction which is "correct" modulo the limitation of random testing. Then, Algorithm 2 is invoked to validate $\mathcal{A}$ (line 12). If it returns true, $\mathcal{A}$ is returned (line 13); otherwise, the process repeats. The details of Algorithm 2 is presented in the subsequent subsection.

Algorithm 2: The validation algorithm

```
Input: abstraction $\mathcal{A} = (S_a, s_a, M, T_a)$; table
       $obs = (P, E, T)$; set $visited$
Output: true iff $\mathcal{A}$ is validated
1  for $\phi \in S_a \setminus \{\bot\}$ and $m \in M$ do
2      if the pair $(\phi, m)$ is not in $visited$ then
3          check the proof obligation $\{\phi\}m\{\psi\}$ using SPF
           where $\psi$ is the disjunction of all $\phi'$ such that
           $(\phi, m, \phi') \in T_a$;
4          if a counterexample is found by SPF then
5              construct a concrete state $s \models \phi$ with the
               counterexample and invoke $m$ on $s$ and obtain
               a concrete state $s'$;
6              if $abs_{Prop}(\{s'\})$ is not in $E$ then
7                  let $tr$ be the shortest trace in $P$ such that
                   $T(tr) = \phi$; update $obs$ with the new
                   mapping $tr \cdot \langle m \rangle \mapsto abs_{Prop}(\{s'\})$;
8                  return false;
9              else
10                 add a transition from $\phi$ to $abs_{Prop}(\{s'\})$
                   labeled with $m$;
11         else
12             add pair $(\phi, m)$ into $visited$;
13 return true;
```

Given that the number of states in $\mathcal{A}$ (and the size of E in the observation table) is bounded by $3^{|Prop|} + 1$, the learning algorithm is always terminating. Furthermore, we argue that $\mathcal{A}$ may be much smaller than this bound in practice. Firstly, variables in a class are often co-related, which is equivalent to say that there are hidden class invariants. Due to those class invariants, often not every abstract state is reachable. For instance, if a hidden class invariant is $v_1 \geq v_2$ and $Prop = \{v_1 \geq 0, v_2 \geq 0\}$, the abstract state $v_1 < 0 \wedge v_2 \geq 0$ is infeasible. Because $\mathcal{A}$ is constructed based on concrete testing results, those hidden class invariants are embedded in $\mathcal{A}$ naturally and hence $\mathcal{A}$ would not contain those infeasible abstract states. Secondly, as mentioned, given a set of concrete states X (reached by the same trace), the abstract state constructed is $abs_{Prop}(X)$, which would effectively collapse many abstract transitions into one. Furthermore, unlike the *L** algorithm, TLV may learn a non-deterministic abstraction, which could be exponentially smaller than its deterministic equivalent. Nonetheless, we admit that the effectiveness of the testing technique may affect the size of the abstraction. We skip the discussion on the complexity of the algorithm as it depends on the effectiveness of the testing techniques. Rather, we show empirically in Section 4 that the learning phase is usually efficient and the generated candidate abstraction usually covers a large portion of the behavior of $\mathcal{C}$.

3.3 Validation

Due to the limitation of random testing, the abstraction learned through testing might not be correct as some behaviors of $\mathcal{C}$ may never be tested. For instance, in the stack example, it is unlikely that we could generate a test case which pushes more than 1024 times and thus the transition $(\top, push, \bot)$ would be missing. However, the abstraction is guaranteed to be accurate (but may not be correct).

LEMMA 3.1. *Algorithm 1 returns an accurate abstraction $\mathcal{A}$.*

Proof (sketch): To prove that $\mathcal{A}$ is accurate, we need to prove that for every transition (ϕ, m, ϕ') in $\mathcal{A}$, there exists a concrete state s such that $s \models \phi$ and invoking m at s would result in a concrete state s' such that $s' \models \phi'$. This is guaranteed by line 8 and 16 in Algorithm 1 which adds a mapping into the observation table such that if $T(tr) = \phi$ and $T(tr \cdot \langle m \rangle) = \phi'$, then there must be a concrete transition from a state satisfying ϕ to a state satisfying ϕ' through invoking m, in both cases. Afterwards, we can prove the lemma based on the construction of $\mathcal{A}$. □

The lemma above states that every transition in $\mathcal{A}$ corresponds to at least one concrete transition. Next, TLV checks if there are missing transitions and if there is none, $\mathcal{A}$ is guaranteed to be an over-approximation at the same time. In the following, we illustrate how the validation algorithm (Algorithm 2) works.

The inputs are the observation table *obs* and the corresponding abstraction $\mathcal{A}$ as well as the set $visited$ which contains pairs of the form (ϕ, m) where ϕ is an abstract state and m is a method name. The set $visited$ stores the successfully discharged proof obligations so far. Every time the algorithm is invoked, for every pair (ϕ, m) of abstract states (exclusive of $\bot$) and methods, TLV checks whether it is in $visited$ (line 2). Intuitively, it is in $visited$ iff TLV has obtained all abstract states which are reachable from ϕ by invoking m. If it is not in $visited$, TLV generates a proof obligation $\{\phi\}m\{\psi\}$ where ψ is the disjunction of all abstract states which are reachable from ϕ through m in $\mathcal{A}$ (line 3). The proof obligation is discharged using symbolic execution, i.e., with the help of Symbolic PathFinder (SPF [41]), as explained in the following.

In a nutshell, given a Java program, SPF executes the code symbolically so as to see whether there is an assertion violation. If an assertion violation is possible, SPF generates a counterexample, which consists of the valuation of input variables and a path condition that lead to the assertion violation. We refer interested readers to work [41] for details on SPF. We instead present how the proof obligation is encoded as an assertion violation checking problem. The first step of the encoding is to syntactically transform the method m such that all relevant instance variables become parameters of the method. Next, TLV instruments the modified method with the required pre-condition ϕ and post-condition ψ. The following illustrates how the instrumentation is done systematically.

```
if (φ) {
  try { body of method m; }
  catch (Exception e) {
    assert false if exception is not in ψ;
  } finally { assert(ψ); }
}
```

TLV first encloses the original method body with a try-catch-finally block to catch all exceptions. The try block contains the method body of m. If $\bot$ logically implies ψ (i.e., $\mathcal{A}$ suggests that exception might be the result when we invoke method m with pre-condition ϕ), the try block contains no assertion; otherwise, it contains the assertion $assert(false)$. Thus, if an exception is not supposed to occur, then the occurrence of an exception would lead to an assertion failure. The finally block contains the assertion $assert(\psi)$ which asserts the post-condition. Next, TLV encloses the try-catch-finally block with an if-conditional block. The condition is set to be the pre-condition ϕ so that SPF checks only symbolic inputs which satisfy the pre-condition. The modified program is then fed to SPF for assertion violation checking.

If no assertion violation is found, the pair (ϕ, m) is added into $visited$ (line 12). Otherwise, using the information returned by SPF, TLV constructs a test case which starts from a concrete state satisfying ϕ and results in a concrete state violating ψ (line 5). Note that in the actual implementation SPF is configured to generate multiple counterexamples at once to reduce the number of SPF invocations. For the stack example, when SPF is used to prove $\{size \geq 0\}push\{size \geq 0\}$, a counterexample is generated which allows TLV to construct a concrete state with $element = 3$ and $size = 1024$. Invoking method $push$ at this concrete state results in state $\bot$ which violates $size \geq 0$. If $abs_{Prop}(\{s'\})$ is in S_a (not a newly discovered abstract state), at line 10, TLV adds a new transition from ϕ to $abs_{Prop}(\{s'\})$. If the abstract state $abs_{Prop}(\{s'\})$ was unreachable previously, at line 7, TLV updates the observation table with a new mapping: $tr \cdot \langle m \rangle \mapsto abs_{Prop}(\{s'\})$ where tr (i.e., the shortest trace which reaches ϕ) is a representative of all traces reaching ϕ. With the new abstract state, the observation table is no longer closed and therefore Algorithm 2 returns false (line 8) and TLV will execute the learning algorithm again to obtain another candidate. The idea is that we always first rely on testing to discover some of the states and transitions inexpensively. *Note that executing the learning algorithm again does not invalidate Lemma 3.1 as we show in the following that $\mathcal{A}$ remains accurate during the validation algorithm.* The validation algorithm returns true when every pair (ϕ, m) is in $visited$(line 13). The following theorem establishes the correctness of TLV.

THEOREM 3.2. *When the validation algorithm (Algorithm 2) terminates, $\mathcal{A}$ is a correct and accurate abstraction of $\mathcal{C}$.*

Proof (sketch): According to Lemma 3.1, $\mathcal{A}$ is accurate before the validation algorithm starts, i.e., for every abstract transition (ϕ, m, ϕ') in $\mathcal{A}$, there is a concrete transition (s, m, s') such that $s \models \phi$ and $s' \models \phi'$. We need to prove that (1) during the validation algorithm, an abstract transition (ϕ, m, ϕ') is added to $\mathcal{A}$ if there is a concrete transition (s, m, s') such that $s \models \phi$ and $s' \models \phi'$; (2) if there is a concrete transition (s, m, s') such that $s \models \phi$ and $s' \models \phi'$, the abstract transition (ϕ, m, ϕ') is in $\mathcal{A}$. (1) is true because new transitions are only introduced at line 10 and (indirectly) at line 7 in Algorithm 2. In both cases, (1) is true as s is obtained from line 5 with a concrete transition. (2) can be proved by contradiction. Assume (s, m, s') is a concrete transition such that $s \models \phi$ and $s' \models \phi'$ and (ϕ, m, ϕ') is not a transition in $\mathcal{A}$. Then there is a proof obligation $\{\phi\}m\{\psi\}$ such that ϕ' does not imply ψ generated at line 3. Assume that SPF works correctly, then the proof must fail, which contradicts the fact all proof obligations must be discharged before the validation algorithm terminates. Thus, we conclude the above theorem is correct. □

In the following, we discuss the complexity of the algorithm. Assume that proving with SPF is terminating, because the number of states in $\mathcal{A}$ is bounded, the validation algorithm always terminates. The number of proof obligations is determined by the number of abstract states in $\mathcal{A}$. In the worst case, it is exponential in the number of propositions in $Prop$. In practice, it is often much less as we show empirically in Section 4. The transitions in $\mathcal{A}$ are discovered through either testing or symbolic execution. The more testing discovers, the less symbolic execution is needed. Because testing is more scalable than symbolic execution, thus by design, TLV minimizes symbolic execution as much as possible. Although $\mathcal{A}$ is correct and accurate, it does not mean that all runs in $\mathcal{A}$ are feasible. For instance, the run $\langle \top, push, \top, push, \bot \rangle$ of the abstraction shown in Figure 3(e) is infeasible. This is essentially due to the phenomenon known as broken traces [25]. We use abstraction refinement to remove such infeasible runs.

3.4 Abstraction Refinement

There are two cases where an abstraction refinement is necessary. One is that the user requires to resolve some non-determinism in the abstraction. The other is to refine the abstraction so as to prune a particular spurious counterexample identified by a model checker. In the following, we explain the latter first and show that the two cases can be solved in the same way.

The abstraction generated after the validation phase is subject to verification techniques like model checking. Assume that the property to be verified is a safety property (e.g., a bounded LTL formula constituted by propositions on instance variables in $\mathcal{C}$). Because the abstraction is guaranteed to be correct, if model checking based on $\mathcal{A}$ concludes there is no counterexample, then the same property is satisfied by $\mathcal{C}$. If a counterexample is identified, we need to check whether it is spurious. If it is spurious, $\mathcal{A}$ must be refined to exclude the spurious counterexample. In the following, we show that a new predicate can be generated based on the information TLV gathered during the learning and validation process. We remark that finding the optimal refinement is known to be hard [14] and is not our goal.

Recall that by assumption, in the setting of verifying a temporal logic formula, $Prop$ contains all propositions in the formula. Let $\langle \phi_0, m_0, \phi_1, m_1, \cdots, \phi_k, m_k, \phi_{k+1} \rangle$ be the spurious counterexample, which is a finite run of $\mathcal{A}$ (as this property is a safety property). Because this run is spurious, it must be broken at some abstract state ϕ_i where $i \leq k$, i.e., invoking m_i at a reachable (from the concrete initial state) state satisfying ϕ_i never results in a state satisfying certain required constraint ϕ_{i+1} [25]. The idea is that if we are able to find a new predicate which could separate those concrete states (abstracted as ϕ_i) which, after invoking m_i, would result in a state satisfying ϕ_{i+1} from those would result in a state violating ϕ_{i+1}, then we can construct a new abstraction (with the new predicate) to rule out this spurious counterexample. For instance, in the stack example shown in Section 2, the spurious counterexample is: $\langle size \geq 0, push, size \geq 0, pop, \bot \rangle$. It is sufficient to rule out the run if we could find a predicate separating those concrete states associated with abstract state $size \geq 0$ into two groups: one resulting in $\bot$ after *pop* and the other resulting in $size \geq 0$ after *pop*. Thus, the problem is to find a classifier for two sets of states, which can be solved using a machine learning based approach [44, 49].

In the case of resolving a non-determinism (as requested by the user), by definition, we have one abstract state, at which calling the same method would result in two different abstract states. Thus, the task of resolving the non-determinism is similarly to find a classifier for two sets of states at the abstract state. In the following, we briefly explain how Support Vector Machines (SVMs) [42] is used to find the classifier.

During the process of generating the abstraction, TLV associates a set of concrete states for each abstract state, which can be partitioned into two groups accordingly. For instance, in the stack example above, one group contains stack objects with $size \geq 1$ (for which there is no exception after *pop*) and the other contains a stack object with $size = 0$ (for which an exception occurs after *pop*). With these two groups (say X and Y), TLV tries to identify a classifier. Formally, a classifier for X and Y is a proposition ω such that for all $x \in X$, x satisfies ω and for all $y \in Y$, and y does not satisfy ω. TLV finds the classifier automatically based on techniques developed by machine learning community, e.g., SVM. As long as X and Y are linearly separable, SVM is guaranteed to find a classifier (i.e., a hyperplane) separating X and Y. Furthermore, there are usually more than one classifiers. In this work, TLV favors the *optimal margin classifier* [44] if possible. This separating hyperplane could be seen as the strongest witness why the two groups are different.

In order to use SVM to generate classifiers, each element in X or Y 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 feedback the SVM results to the original program. We leverage our earlier approach [49] to 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.

Recall that one group contains stack objects with $size = 1$ and the other contains a stack object with $size = 0$. TLV first extracts two sets of feature vectors from the two groups using the first level features (i.e., features which can be accessed using the stack object and no other references) in the graph, i.e., $isNull$ and $size$. The first set of feature vectors is $\{\langle 0, 1 \rangle\}$ where $\langle 0, 1 \rangle$ denotes the stack object is not null (i.e., 0 means that $isNull$ is false) and its variable $size$ is of value 1. The second set is $\{\langle 0, 0 \rangle\}$. Next, SVM finds a classifier $2 * size \geq 1$. Notice that if SVM fails to find a linear classifier based on the two sets of feature vectors, TLV constructs two new sets by using numerical values from next level in the graph (i.e., $isNull$ for $elements$ and $length$ of $elements$, and the actual data in the array) and tries SVM again. The heuristic that we look for a classifier level-by-level is based on the belief that 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.

4. EVALUATION

TLV (available at [48]) is implemented with about 35K lines of Java code. We use Eclipse JDT to analyze and instrument Java source code, e.g., for generating modified programs for symbolic execution. We use SPF [39, 41] for symbolic execution because it supports features (such as assertion checking) which are necessary in our approach. The experimental results are collected on a 64-bit Ubuntu 14.04 PC with a 3.10GHz Intel Core i3 processor and 4GB memory. For the learning phase, we generate 4 concrete values for each argument of the method in an abstract trace and each learning iteration is set to timeout in 1 minute. We evaluated TLV to answer three research questions.

RQ1: How effective and scalable is TLV?

The answer to this question depends not only on the capability of TLV but also on SPF. Thus, we answer the question by applying TLV to two groups of Java classes, one containing relatively simple classes which we could get useful results from SPF and the other containing real-world classes which are beyond the capability of SPF. The idea is to show that TLV is able to generate correct and accurate abstraction efficiently if the symbolic execution engine is working as hoped, and TLV is able to generate meaningful abstractions (without soundness guarantee) for large programs even when SPF fails to provide any support.

The first group contains 12 Java classes. In particular, classes ALTBIT, FLIGHTRULE, INTMATH, SIGNATURE, SOCKET and STREAM were used in the evaluation of X-PSYCO tool [28]; SERVERTABLE and LISTITR are from the work [1]; BANKACCOUNT is from the work [50] and EWALLET and PAYAPPLET are adopted from Java Card applets [30]. To determine the level of abstraction, for each class, we set the initial predicates to be those collected from the source code, e.g., conditions in "if" and "for" statements. In our experience, those predicates would often allow us to quickly get some idea of the class behavior. The set of predicates

Table 1: Statistics on TLV abstracting the classes, where N.A. stands for not available

	Inputs			Learning			Initial Abs		Validation		Final Abs		Memory (MB)
Class	LOC	$\#M$	$\#Pred$	$\#MQ$	$\#CQ$	$T_L(s)$	$\#S$	$\#T$	$T_V(s)$	$\#Prof$	$\#S$	$\#T$	TLV
AltBit	60	2	3	22	1	0.4	5	11	7.0	8	5	11	104
BankAccount	40	2	3	22	1	0.7	3	8	4.7	4	3	8	105
BoundedStack	45	2	3	8	2	0.1	2	3	7.0	2	3	7	104
FlightRule	50	3	1	10	1	0.2	2	3	2.7	3	3	8	104
IntMath	500	8	1	5209	1	60.3	2	16	5.0	8	2	16	544
ListItr	40	5	4	320	1	1.3	6	36	25.7	25	6	47	291
ServerTable	90	6	6	5485	1	60.2	6	42	29.3	30	6	42	712
Socket	200	7	10	6203	1	60.2	13	102	167	168	25	219	559
Stream	180	4	3	41	1	0.6	3	9	7.0	8	3	9	104
Signature	50	5	4	61	1	0.4	4	15	13.1	15	4	15	104
eWallet	90	4	5	33	1	0.4	2	5	7.5	8	3	13	169
PayApplet	100	5	6	31	1	0.3	2	5	29.8	25	6	29	104
SocketReal	1660	7	6	1807	1	60.6	13	85	N.A.	N.A.	13	85	381
JavaMailReal	2000	5	2	21	1	14.6	3	10	N.A.	N.A.	3	10	357
StreamReal	180	4	4	2725	1	60.2	4	14	N.A.	N.A.	4	14	454

for each of the above target classes can be accessed at our website [48]. We acknowledge that these classes are relatively small because most of the above-mentioned approaches (like TLV) are limited to the capabilities of symbolic execution.

To show that TLV is useful even without the validation phase, we collect a second group of classes, containing real-world programs, and apply TLV to generate abstraction. In particular, SocketReal is the *java.net.Socket* class defined in JDK 7 (with >1.6K LOC in the class and >20K LOC in the referenced library); JavaMailReal is the *com.sun.mail.smtp.SMTPTransport* class defined in the JavaMail library (version 1.5.2, with 2K LOC in the class and >45K LOC in the referenced library). StreamReal is the JDK 7 class *java.io.PipedOutputStream* class (with 180 LOC in the class and >3K LOC in the referenced library). These programs are un-modified other than that we set the first two programs to connect to a local socket server and mail server for testing purpose (as did in TAUTOKO [16, 17]). They either use Java Native Interface or contain reference type fields and parameters, which are not supported by SPF. To determine the level of abstraction, we manually inspect the classes and collect predicates which we believe are associated with the class invariants. Based on the abstractions learned by TLV we confirm that those are indeed class invariants.

The statistics on the experiments with these two groups of classes are shown in Table 1. Column #M shows the number of methods used for abstraction; Column #$Pred$ is the number of propositions for each class (excluding the one on whether the state is $\perp$). We collect the number of membership queries (column #MQ), and the number of candidate queries (column #CQ), and the total time (column T_L) to learn the initial abstraction. The statistics for the validation phase are the number of proof obligations (column #$Prof$, i.e., the size of $visited$ in the validation algorithm) and the time used in the validation phase (column T_V). A closer look shows that T_L is dominated by the time spent on maintaining the observation table (so as to make it memory efficient by merging abstract states/transitions). On the contrary, running the test cases only takes a very small portion of the time. T_V includes the interprocess communication time and each invocation of SPF often takes less than one second. The instrumentation and compilation time in the validation phase are negligible. For all classes, we manually confirm the correctness and accuracy of the generated abstractions. It is shown that for all classes, TLV generates the abstraction in minutes. Furthermore, the overall time is dominated mostly by the validation algorithm (for 10 out of 12 cases) for the first group of classes. For all classes, the peak memory consumption for TLV is 712 MB and thus TLV is reasonably memory efficient.

Comparison with other tools To the best of our knowledge, TLV is the only tool which combines testing, learning and validation for abstracting Java classes. There are two existing tools on predicate abstraction of Java programs: J2BP [38] and X-Psyco [28]. Our investigation of J2BP shows that it generates abstractions for a Java program with a "main" method and not for Java classes. Furthermore, it does not support random numbers or symbolic inputs, we are unable to write a driver program so that J2BP can be used to abstract a Java class. X-Psyco is designed for generating an interface specification. It discovers predicates (through symbolic execution) which are constituted by method parameters to specify constraints which must be satisfied in order to invoke the method. X-Psyco assumes that only propositions on method parameters and return values are relevant, which implies that X-Psyco is targeting completely different programs from TLV. Thus, we conclude that X-Psyco and TLV are complementary but incomparable. In addition, there is one ongoing effort by the JPF team [19] and a previously reported tool [1], which is not available any more. Besides tools for predicate abstraction, there are tools of learning models of Java classes, among which we identify the TAUTOKO tool [16, 17] to be bearing a similar goal as TLV. Thus, in the following, we compare TLV with TAUTOKO[1] in the context of answering the above research question.

TAUTOKO first uses ADABU [18] to generate a model for each test case in the user-provided test suite and combines these models into an initial model. It then mutates existing test cases to generate more tests cases from the initial model and combines models for new test cases with the initial model to generate an enriched model. For fairness, we use the test cases generated by TLV in the learning phase as the input test suite for TAUTOKO. For the predicates, TAUTOKO is limited to predicates generated with a set of abstraction templates over instance variables of the given class, while TLV is more flexible. Thus, we set the predicate used in TLV to be those used in TAUTOKO. We compare TLV and TAUTOKO by applying them to the first group of classes. Notice that TAUTOKO has trouble obtaining models for the second group of classes for various reasons, i.e., TAUTOKO cannot handle SocketReal as TAUTOKO does not support Java 7; TAUTOKO does not generate models for StreamReal because it does not instrument classes in *java.io* package; TAUTOKO fails to execute the test suite for JavaMailReal. The statistics for the models generated by TAUTOKO and TLV are shown in Table 2, which shows the number of states (column #S) and transitions (column #T) discovered by TLV and TAUTOKO, respectively. In addition, #T_e denotes the numbers of transitions to state $\perp$, which is a useful

[1]We use the version of TAUTOKO reported in [17] as the implementation reported in their later work [16] is not available.

Table 2: Comparing TLV with TAUTOKO on abstraction

Class	TLV				TAUTOKO			
	$\#S$	$\#T$	$\#T_e$	$T(s)$	$\#S$	$\#T$	$\#T_e$	$T(s)$
AltBit	5	11	8	7.6	4	5	2	37.2
BankAccount	5	19	8	141.2	4	12	0	1817
BoundedStack	3	7	2	4.4	3	5	1	14.2
FlightRule	3	8	3	6.5	2	3	2	12.6
IntMath	2	16	8	5.9	1	6	0	6.4
ListItr	6	47	19	27.0	3	10	2	20.2
ServerTable	12	118	52	68.5	7	27	10	244.2
Socket	25	219	146	174.0	2	8	1	475.8
Stream	3	9	2	7.9	3	7	1	9.7
Signature	4	15	5	14.4	4	15	5	18.4
eWallet	3	13	8	8.2	2	2	1	2.4
PayApplet	6	29	20	15.3	2	3	2	12.5

Table 3: Comparing TLV's abstraction and manual abstraction

Class	Testing and Learning			Manually Constructed		
	$\#S$	$\#T$	$\#T_e$	$\#S$	$\#T$	$\#T_e$
JavaMailReal	3	10	2	3	10	2
StreamReal	4	14	6	4	14	6
SocketReal	13	85	62	13	85	62

metric [17]. The results show that TLV always generates more accurate models (because more valid states and transitions are generated) than TAUTOKO. The time statistics (column T) show that TLV is often more efficient than TAUTOKO, especially when the number of test cases is large.

RQ2: How effective are testing and learning?

This question evaluates TLV's underlying assumption, i.e., testing and learning could effectively reduce the effort on symbolic execution. For the second group of classes, the answer to the question would determine how much behavior the abstraction contains and thus how useful it is, since the symbolic execution engine is helpless for these classes. For the first group of classes, this question is answered by measuring the percentage of abstract states/transitions which are discovered in the learning phase. In particular, we compare the initial candidate abstraction (column Initial Abs) which is generated based on testing and learning only and the final abstraction (column Final Abs). We collected the respective number of states (column #S) and number of transitions (column #T). It can be observed that for most of the classes in the first group (11 out of 12), most of the states (96%) and transitions (94%) are discovered during learning based on the test cases, which suggests that our underlying assumption is often valid. On the other hand, there is only one class (PayApplet) where testing is shown to be ineffective in discovering the behavior (only 17% of the transitions are discovered by testing), which evidences that a validation phase is indeed necessary. A closer look at the class shows that only the method "setKey" leads to a state (which is then connected to a number of other states) from the initial state. Furthermore, this transition can only happen when a particular integer parameter value is passed in (there is an "if" statement with condition $size = (DES\_KEY\_SIZE + ID\_SIZE)$). TLV did not generate the particular integer value and thus missed many states. We expect this could be more often with larger and more complicated programs, which might pose a thread to TLV. On the other hand, by comparing the T_L with T_V and contrasting T_L with the number of transitions discovered during testing, it is evident that testing discovers abstract states/transitions much cheaper than symbolic execution and therefore it is wise to start with testing and learning.

For the second groups of classes, we compare the generated abstractions with those constructed manually using the same set of predicates. The results in Table 3 show that TLV can learn all the

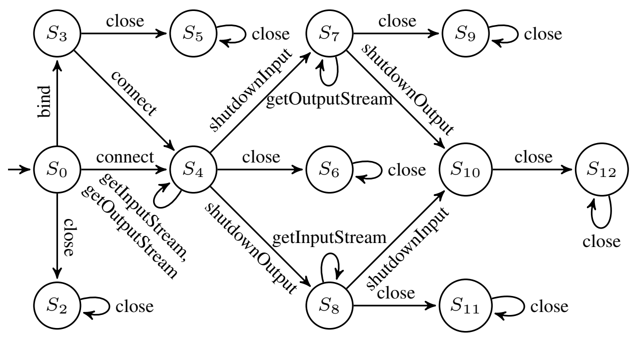


Figure 6: Abstraction for *java.net.Socket* class with the following predicates: $closed = T, created = T, bound = T, shutIn = T, shutOut = T, connected = T$**, where the error state** S_1 **and all transitions to it are omitted for brevity.**

states and transitions of these classes with regard to the given set of predicates. In general this is hard to achieve. However, for these classes whose methods have no or few parameters, learning-guided testing, as implemented in TLV, is effective at discovering most of the behaviors systematically. In the following, we present some details on the SocketReal case. The abstraction generated by TLV is shown in Figure 6 with the 6 predicates used for abstraction. The predicates are obtained from suspected class invariants, e.g., $connected = T$ implies $bound = T$ and $bound = T$ implies $created = T$. Thus, part of the goal of using TLV to generate the abstraction is to check whether these are indeed class invariants. The abstraction learned by TLV contains (only) 12 valid states plus the error state. In addition to those shown in Figure 6, there is a transition from S_0 to S_1 labeled with $bind$ which occurs when the port number (e.g., 3) for binding is reserved and thus the method call results in permission violation exception. Note that other transitions to the error state are omitted for brevity. We confirm that the abstraction is correct and accurate by manually discharging all the proof obligations. Based on the abstraction, we can easily confirm that the suspected class invariants are indeed invariants since all states in the abstraction satisfy them.

RQ3: Does TLV learn good predicates automatically?

We remark that this question is best answered with specific properties to be verified and specific spurious counterexamples returned by a verification engine. We have integrated a model checker [46] into TLV, which makes TLV a fully automatic Java model checker based on abstraction and refinement (by learning new predicates).

In order to answer this question, we evaluate TLV's capability of refinement in the following setting. We assume TLV is being used to verify a Java class against a precise *deterministic* specification on when an exception occurs in the class, i.e., a 'typestate' [45]. Note that such a specification often involves predicates on instance variables. Thus, TLV is first used to construct an abstraction with one proposition $true$. The result is an abstraction containing two states: $true$ and $\perp$. Next, TLV refines the model by discovering new predicates which would show exactly when an exception occurs (and thus rules out spurious counterexamples found by verifying the abstraction against the given specification). We show that TLV eventually finds the right predicates based on testing results and learning.

The results are shown in Table 4 (note that not all of the classes have a nontrivial stateful typestate). Note that for class PayApplet, instead of $state = 0$, SVM generates two predicates $state \geq 0$ and $state \leq 0$ and uses their conjunction to obtain the same result. We remark in this setting, TLV solves the prob-

Table 4: Statistics for automatic predicate generation

Class	Time (s)	Mem (MB)	Predicates
BOUNDEDSTACK	77.5	361	$size \geq 0, size \geq 1024$
SIGNATURE	43.0	124	$state \leq 0, state \leq 1, state \leq 2$
PAYAPPLET	349.4	379	$state \geq 0, state \leq 0, size \geq 16, size \leq 16, value + state > 1$

lem of synthesizing a stateful typestate, as studied in TzuYu [49]. Different from TzuYu [49], the typestate generated by TLV is guaranteed to be correct (as well as accurate).

Limitations TLV has two main limitations. First, because TLV employs symbolic execution for abstraction validation, it inherits the limitation from symbolic execution engines, e.g., limitations in handling programs with loops or complex data structures. We optimistically believe that TLV (like previous approaches) would handle larger programs with the rapid development of program verification techniques. Second, because TLV relies on random testing to discover behaviors, the performance of TLV would suffer if the program contains many behaviors which are hard to be explored by random testing (in which case, TLV constructs the abstraction solely based on symbolic execution).

5. RELATED WORK

To the best of our knowledge, TLV is the first to combine testing, learning, and validation for program abstraction. TLV is inspired by research on predicate abstraction [1, 9], specification mining [22, 28, 49], testing for predicate-coverage [4], etc. TLV is a generalization of TzuYu [49], which is designed to learn a typestate for a Java class. On one hand, TzuYu and TLV both rely on random testing and learning to generalize models from concrete tests. On the other hand, TzuYu learns only typestates which has only two states ($\top$ and $\bot$), whereas TLV can handle more predicates; TzuYu's learning algorithm is a direct adoption of the *L** algorithm, whereas TLV's learning algorithm is designed to maximize predicate-coverage [4]. More importantly, TzuYu provides neither correctness nor accuracy guarantee of the typestate, whereas TLV does.

Alur *et al.* [1] construct the interface specification of a Java class based on active learning and use a model checker as the teacher; the Sigma* tool [9] learns the symbolic input/output relation for a transducer program by using symbolic execution to find new program paths; the PSYCO tool [22] uses symbolic execution to answer both membership queries and candidate queries for learning a specification of a class. The X-PSYCO tool [28] combines testing and symbolic execution to learn a symbolic abstraction of program behavior. Concrete test inputs in X-PSYCO are generated with symbolic execution. Although learning is applied in TLV and these approaches, TLV is built on the idea of "test as much as we can" and thus avoids techniques like model checking or symbolic execution as much as possible. In addition, TLV has a clear target level of abstraction which is often unclear in the above approaches.

The IDISCOVERY tool [50] combines dynamic invariant inference (DAIKON [20]) with symbolic execution to generate more precise invariants. Invariants generated by IDISCOVERY are not targeted for a specific usage context, whereas TLV is designed to generate abstraction at certain level of abstraction for a particular usage context. As mentioned earlier, TLV bears a similar goal as the TAUTOKO tool [16, 17]. Different from TAUTOKO, TLV uses symbolic execution to discover more states and transitions and to provide correctness guarantee. Furthermore, TLV's abstraction is catered for specific usages and has a targeted level of abstraction.

Existing approaches on building finite state models [7, 18, 32, 34, 36] use similar state abstraction strategies as used in TLV. The STRAWBERRY tool [7] mines behavior protocols concerning usage of a web service. ADABU [18] generates invariants from concrete execution traces of Java classes with a set of fixed invariant templates. ReAjax [34] uses a very similar way as ADABU to generate the abstract model for the Document Object Model of an Ajax web pages. KrKa *et al.* [32] use DAIKON to generate a set of possible state invariants and then use SMT solvers to decide the feasibility of possible states (combinations of invariants) and transitions in the abstract model. Revolution [36] mines state based behavior model for systems whose behaviors may evolve with time.

TLV uses automata learning and testing to construct the initial abstraction. Similar ideas have been used to generate models for legacy systems [29, 35] and bug detection [40]. They use *L** to learn Deterministic Finite Automata or Mealy Machines, whereas the learning algorithm used in TLV learns a Non-deterministic Finite Automaton (NFA). TLV requires the source code of the class under test to generate abstraction while some techniques [7, 21] treat the System Under Test (SUT) as black-box, thus they use only the external visible values for state abstraction. Ghezzi *et al.* [21] use behavior equivalent model for finite data container to generate an abstract model for a given Java class with a test suite.

*L**-based learning algorithms have been applied not only to learning specification from source code [1, 9, 22, 28, 49], but also to model checking of programs [24] as well as security domain [12, 13]. TLV's learning approach learns NFAs, whereas *L** learns only DFA which is limited for programs with data variables. TLV is related to work on extending the *L** algorithm to learning NFA [8] or other finite state models [6, 10, 33, 43]. Bollig *et al.* [8] extend the *L** to learn a non-deterministic Residual Finite State Automata (RFSA) for a deterministic SUT. Berg *et al.* [6] extend *L** to learn a symbolic Mealy Machine. Shahbaz and Groz [43] introduce a direct Mealy Machine learning algorithm which handles counterexamples returned by the teacher much more efficiently. Lorenzoli *et al.* [33] propose the *GK-tail* algorithm to generate an Extended Finite State Automata. Cassel *et al.* [10] extend *L** to learn symbolic register automata with tree queries.

TLV proposes an alternative approach for predicate abstraction. Graf and Saïdi [23] invent predicate abstraction. Ball *et al.* [5] propose predicate abstraction for C programs. There are many extensions such as lazy abstraction by Henzinger *et al.* [27] and the work by Arie *et al.* [26]. The J2BP tool [38] and its variant Abstract Pathfinder [19, 31] are the first to do predicate abstraction for Java programs. All these work relies solely on SMT solvers to compute the abstraction, whereas TLV combines testing, learning, and validation. Lastly, TLV can be viewed as an approach to testing Java classes for predicate-coverage [4]. Visser *et al.* [47] observed that predicate-coverage is harder to achieve than block-coverage for testing. In a way, TLV aims to provide complete predicate-coverage by integrating testing, learning, and validation.

6. CONCLUSION

In short, we proposed an approach named TLV, which combines testing, learning, and validation in order to automatically generate predicate abstraction of Java classes and to automatically refine the abstraction if necessary. TLV generates accurate and correct abstractions efficiently. As for future work, we are experimenting different testing strategies and machine learning algorithms to further improve TLV's performance.

7. REFERENCES

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. In *POPL*, pages 98–109, 2005.

[2] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[3] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *NDSS*, 2013.

[4] T. Ball. A Theory of Predicate-Complete Test Coverage and Generation. In *FMCO*, pages 1–22, 2004.

[5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI*, pages 203–213, 2001.

[6] T. Berg, B. Jonsson, and H. Raffelt. Regular Inference for State Machines Using Domains with Equality Tests. In *FASE*, pages 317–331, 2008.

[7] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic Synthesis of Behavior Protocols for Composable Web-services. In *ESEC/FSE*, pages 141–150, 2009.

[8] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style Learning of NFA. In *IJCAI*, pages 1004–1009, 2009.

[9] M. Botinčan and D. Babić. Sigma*: Symbolic Learning of Input-output Specifications. In *POPL*, pages 443–456, 2013.

[10] S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Learning Extended Finite State Machines. In *SEFM*, pages 250–264, 2014.

[11] S. Chaki, E. M. Clarke, O. Grumberg, J. Ouaknine, N. Sharygina, T. Touili, and H. Veith. State/Event Software Verification for Branching-Time Specifications. In *IFM*, pages 53–69, 2005.

[12] C. Y. Cho, D. Babi ć, E. C. R. Shin, and D. Song. Inference and Analysis of Formal Models of Botnet Command and Control Protocols. In *CCS*, pages 426–439, 2010.

[13] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *USENIX Security Symposium*, 2011.

[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169, 2000.

[15] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[16] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically Generating Test Cases for Specification Mining. *IEEE Trans. Software Eng.*, 38(2):243–257, 2012.

[17] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating Test Cases for Specification Mining. In *ISSTA*, pages 85–96, 2010.

[18] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining Object Behavior with ADABU. In *WODA*, pages 17–24, 2006.

[19] J. Daniel, P. Parízek, and C. S. Păsăreanu. Predicate Abstraction in Java Pathfinder. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, 2014.

[20] 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. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[21] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing Intensional Behavior Models by Graph Transformation. In *ICSE*, pages 430–440, 2009.

[22] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic Learning of Component Interfaces. In *SAS*, pages 248–264, 2012.

[23] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83, 1997.

[24] A. Groce, D. Peled, and M. Yannakakis. AMC: An Adaptive Model Checker. In *CAV*, pages 521–525, 2002.

[25] A. Gupta and E. M. Clarke. Reconsidering CEGAR: Learning Good Abstractions without Refinement. In *ICCD*, pages 591–598, 2005.

[26] A. Gurfinkel, S. Chaki, and S. Sapra. Efficient Predicate Abstraction of Program Summaries. In *NFM*, pages 131–145, 2011.

[27] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL*, pages 58–70, 2002.

[28] F. Howar, D. Giannakopoulou, and Z. Rakamarić. Hybrid Learning: Interface Generation Through Static, Dynamic, and Symbolic Analysis. In *ISSTA*, pages 268–279, 2013.

[29] H. Hungar, T. Margaria, and B. Steffen. Test-based Godel Generation for Legacy Systems. In *ITC*, pages 150–159, 2003.

[30] IBM. Java Card Technology. `http://www.oracle.com/technetwork/java/embedded/javacard/overview/default-1969996.html`, May 2014.

[31] A. Khyzha, P. Parízek, and C. S. Păsăreanu. Abstract Pathfinder. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, 2012.

[32] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using Dynamic Execution Traces and Program Invariants to Enhance Behavioral Model Inference. In *ICSE*, pages 179–182, 2010.

[33] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *ICSE*, pages 501–510, 2008.

[34] A. Marchetto, P. Tonella, and F. Ricca. State-Based Testing of Ajax Web Applications. In *ICST*, pages 121–130, 2008.

[35] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient Test-based Model Generation for Legacy Reactive Systems. In *HLDVT*, pages 95–100, 2004.

[36] L. Mariani, A. Marchetto, C. Nguyen, P. Tonella, and A. Baars. Revolution: Automatic Evolution of Mined Specifications. In *ISSRE*, pages 241–250, 2012.

[37] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *ICSE*, pages 75–84, 2007.

[38] P. Parízek and O. Lhoták. Predicate Abstraction of Java Programs with Collections. In *OOPSLA*, pages 75–94, 2012.

[39] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *ASE*, pages 179–180, 2010.

[40] M. Pradel and T. R. Gross. Leveraging Test Generation and Specification Mining for Automated Bug Detection Without False Positives. In *ICSE*, pages 288–298, 2012.

[41] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining Unit-level Symbolic Execution and System-level

Concrete Execution for Testing Nasa Software. In *ISSTA*, pages 15–26, 2008.

[42] B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors. *Advances in Kernel Methods: Support Vector Learning*. MIT Press, 1999.

[43] M. Shahbaz and R. Groz. Inferring Mealy Machines. In *FM*, pages 207–222, 2009.

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

[45] R. E. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[46] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 709–714, 2009.

[47] W. Visser, C. S. Pasareanu, and R. Pelánek. Test Input Generation for Java Containers Using State Matching. In *ISSTA*, pages 37–48, 2006.

[48] H. Xiao. TLV hosting site. `http://bitbucket.org/spencerxiao/tlv-fse2015`, Jan 2015.

[49] H. Xiao, J. Sun, Y. Liu, S.-W. Lin, and C. Sun. TzuYu: Learning Stateful Typestates. In *ASE 2013*, pages 432–442, 2013.

[50] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven Dynamic Invariant Discovery. In *ISSTA*, pages 362–372, 2014.