Adversarial Specification Mining

HONG JIN KANG, School of Information Systems, Singapore Management University
DAVID LO, School of Information Systems, Singapore Management University

There have been numerous studies on mining temporal specifications from execution traces. These approaches learn finite-state automata (FSA) from execution traces when running tests. To learn accurate specifications of a software system, many tests are required. Existing approaches generalize from a limited number of traces or use simple test generation strategies. Unfortunately, these strategies may not exercise uncommon usage patterns of a software system. To address this problem, we propose a new approach, adversarial specification mining, and develop a prototype, DICE (Diversity through Counter-Examples). DICE has two components: DICE-Tester and DICE-Miner. After mining Linear Temporal Logic specifications from an input test suite, DICE-Tester adversarially guides test generation, searching for counterexamples to these specifications to invalidate spurious properties. These counterexamples represent gaps in the diversity of the input test suite. This process produces execution traces of usage patterns that were unrepresented in the input test suite. Next, we propose a new specification inference algorithm, DICE-Miner, to infer FSAs using the traces, guided by the temporal specifications. We find that the inferred specifications are of higher quality than those produced by existing state-of-the-art specification miners. Finally, we use the FSAs in a fuzzer for servers of stateful protocols, increasing its coverage.

CCS Concepts: • **Software and its engineering → Software reverse engineering**;

Additional Key Words and Phrases: specification mining, search-based test generation, fuzzing

ACM Reference Format:
Hong Jin Kang and David Lo. 2020. Adversarial Specification Mining. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2020), 41 pages. https://doi.org/10.1145/3424307

1 INTRODUCTION

In an ideal world, software systems and their APIs should be clearly documented to prevent misuse and bugs. In practice, software systems are usually released without documentation and existing documentation may become out of date as the software system evolves [67]. The lack of specifications causes difficulties in program comprehension, and the misuse of APIs has been recognized as a leading cause of software bugs and vulnerabilities [21, 45].

To address the lack of specifications, researchers have proposed many techniques to automatically infer specifications and usage models, frequently in the form of Finite State Automata (FSA). These techniques require execution traces of the software as input. It is assumed that these traces are representative of the software and that all correct behavior are reflected in these traces. Unfortunately, it was found that automatically mined specifications are still inaccurate [37]. One reason for this may be that the traces used to construct these specifications are not representative and are not sufficiently diverse. Other researchers have proposed techniques [9, 14] to determine if

Authors' addresses: Hong Jin Kang, hjkang.2018@phdis.smu.edu.sg, School of Information Systems, Singapore Management University, 80 Stamford Rd, Singapore, 178902; David Lo, School of Information Systems, Singapore Management University, 80 Stamford Rd, Singapore, 178902, davidlo@smu.edu.sg.

1049-331X/2020/1-ART1 $15.00
https://doi.org/10.1145/3424307

enough traces have been seen, but these techniques only consider the traces that have already been seen and metrics are computed only over the observed traces. They are unable to reason about execution traces which are possible but are uncommon.

Test generation may be a way to generate new tests that specification miners can learn from, and in several studies, researchers have used test generation to mine specifications. Tautoko [15], for example, refines a FSA-based specification by generating tests to cover missing transitions. Deep Specification Miner (DSM) [36] leverages random test generation to produce a large number of traces to learn language models from. Still, as we investigate later in this study, these techniques are not sufficient for producing highly accurate models. DSM relies on randomized test generation, and even when provided with traces of uncommon usage, it is not able to leverage these traces to produce more accurate models. Tautoko relies on methods that reveal the state of an object to detect a state change, which may limit it from working effectively for all types of objects. As such, we hypothesize that existing test generation strategies do not completely address the problem of uncommon usage patterns.

For ensuring that uncommon usage is represented, we propose a process which we term *adversarial specification mining*. In the first phase, we mine specifications from traces collected from running a set of test cases of the software under test. In the second phase, test generation is guided towards the discovery of counterexamples of the mined specifications. In the third phase, a specification miner uses the new counterexamples to construct an accurate model. We developed a prototype, *DICE* (**Di**versity through **C**ounter-**E**xamples). DICE mines FSA models through an adversarial specification mining process. For the purpose of inferring a more accurate model, DICE produces more example execution traces given an initial set of temporal specifications, aiming to find inaccuracies in them. This is done through a search-based test generation process is adversarial to the input specifications, searching for tests that exercise the software under test in ways that the input specifications would not accept as correct usage. DICE contains two main components: **DICE-Tester**, which drives test generation towards uncommon patterns, and **DICE-Miner**, which converts execution traces into a Finite-State Automata (FSA) model. This is the first study that makes use of search-based testing for mining specifications.

DICE-Tester uses a search-based testing framework, Evosuite[25], guiding it towards the generation of counterexamples of the input specifications by representing traces that will falsify the specifications as search goals. The modifications made by DICE-Tester prevents the search algorithm in Evosuite from getting caught in a local optima, enabling Evosuite to efficiently search for counterexamples. We use the DynaMOSA algorithm, introduced in a previous study [50], to allow Evosuite to dynamically select objectives instead of trying to achieve pareto optimality.

To characterize a set of traces, we first mine specifications as properties in Linear Temporal Logic (LTL), a formalism of constraints on event-ordering, that hold on the traces. Prior work has shown the relationship between LTL and specification mining [7, 8, 20, 34, 66], and has applied data mining techniques to infer temporal properties [10, 35]. Still, automatically mined properties are typically not completely accurate. As such, this motivates work on boosting the accuracy of identifying temporal properties. Adversarial specification mining solves this problem by filtering out temporal properties that can be invalidated. In this study, we use six LTL property templates introduced in previous studies [7, 34, 61]. However, we propose a reformulation of three properties, in which we use knowledge of method purity derived from low-cost heuristics and static analysis, to reformulate the properties, tackling shortcomings of the properties described in a recent study [61]. In this work, we use the terms "pure" and "side-effect-free" interchangeably.

As a result of guiding test generation to search for counterexamples, the traces collected by DICE-Tester include uncommon, but correct, usage patterns of the library. To infer an FSA model, we use the traces and temporal properties in a FSA inference algorithm that we propose. We borrow

insights from prior work [17, 33], characterizing the states in an FSA based on the methods that are enabled and which can be invoked from it. We make two observations of limitations in existing model inference algorithms and modify our algorithm to address them. In our evaluation of DICE, we find that the models produced by DICE outperform models from existing specification miners, such as the state-of-the-art specification miner, Deep Specification Miner [36]. Finally, we compare DICE and DSM by using the models they infer in server fuzzing, and we find that the models learned by DICE helps in increasing line and branch coverage on an FTP server.

This paper makes the following contributions:

- We propose the approach of adversarial specification mining. Our prototype of this approach, DICE, produces a Finite-State-Automata from an initial test suite. This process involves two main components, a test generation process identifying counterexamples, and a state-of-the-art FSA specification mining process.
- We are the first to propose the use of an adversarial, search-based test generation process to produce diverse examples to learn from.
- We propose a new specification miner that uses LTL properties, and avoids weaknesses of existing approaches, allowing for the inference of higher-quality Finite State Automata.
- We empirically evaluate DICE against multiple baseline approaches, including state-of-the-art tools.
- We use the state machine output from DICE in a stateful server fuzzer proposed in prior work, and we show that DICE and the learned state machines can help improve fuzzing.

The rest of the paper is organized as follows. Section 2 introduces the background of this study. Section 3 presents DICE in detail. Section 4 empirically compares DICE against other specification mining tools, answering several research questions of this study. Section 5 provides some discussion of our work, including a qualitative evaluation, its practical applicability in server fuzzing, and threats to validity. Section 6 discusses studies related to our work. Section 7 presents our conclusions and future directions of our work

2 BACKGROUND

2.1 Specification Mining

2.1.1 K-tails and its variants. Many specification mining algorithms have been proposed. To infer FSA, specification mining algorithms have to provide abstractions over states, and determine if two traces result in the same state. A classic algorithm that infers a Finite State Automaton from traces is the k-tails algorithm [3]. The k-tails algorithm [3] first uses the input execution traces to build a Prefix Tree Acceptor (PTA). A PTA is a tree-like deterministic finite automaton (DFA) where states are grouped and merged based on the prefix that they share. This automaton is consistent with the input traces and will accept all of them. Next, the algorithm merges states that have the same sequences of invocations in the next k steps. The value of the parameter, k, can vary. This trades off precision and recall; a small value of k results in more spurious merges while a large value of k leads to lower generality.

Studies have extended the traditional k-tails algorithm. These studies often keep the first step of the original k-tails algorithm, using a PTA to create a tree-like automaton that directly represents the input traces. These algorithms thus inherit the assumptions made by k-tails in its first step, that states with the same prefix are equivalent, typically only modifying the second step of the k-tails algorithm, changing the equivalence criteria of states before merging them.

Lo et al. [41] propose to mine temporal rules that hold over the input traces and prevent any merge that will result in a violation of the rules. Lorenzoli et al. [42] introduce GK-tail, which mines extended FSA where transitions are labelled not only with method calls, but includes parameter

values. They introduce different merging criteria, including criteria that do not require exact matches of the transitions, and allow for more general conditions of the parameter values. Krka et al. [33] introduce multiple algorithms in their work, including SEKT, which extends k-tails by adding another condition for equivalence: States are merged only if they correspond to the same abstract state, which are defined by the invariants extracted by Daikon. Le et al. [36] propose a deep-learning based approach, Deep Specification Miner (DSM), to determine if a set of states are equivalent. They train a Recurrent Neural Network-based model to produce features characterizing each state in a high-dimensional space. After clustering the states in this space, states are merged according to the clusters they belong to. Therefore, each cluster is mapped to a single state in the output FSA. In this study, states are characterized by a feature vector built for each state, which includes the likelihood of each possible transition label based on their prefix.

2.1.2 State abstraction. While k-tails and its variants combine states based on their prefixes and an equivalence criteria, other approaches have proposed other methods to infer the states in an FSA. de Caso et al. [17] propose CONTRACTOR, which uses program invariants to characterize states of an FSA based on the *enabledness* of methods. A method is enabled if the invariants of the state hold. States are thus a combination of enabled methods, where the pre-conditions of the methods are consistent with one another. CONTRACTOR was proposed as a method to validate pre- and post-conditions specifications by presenting a state machine abstraction of the specifications. The finite state machine help in revealing potential inaccuracies among the pre- and post-conditions. Constructing all possible combinations of enabledness of the methods result in number of states exponential to the number of methods. To avoid this state space blowup, CONTRACTOR models the dependencies between method enabledness to reduce the number of states. Afterwards, only the states reachable from the initial state are retained. Krka et al. [33] enhance the CONTRACTOR model by proposing CONTRACTOR++, filtering invariants inferred by Daikon [22] and including the output value of method invocations in the labels of transitions.

Finally, approaches such as ADABU [16] and Tautoko [15] identify a set of inspectors for each class. Inspector methods are heuristically identified based on their return type (not void), a lack of parameters, and the lack of side-effects. Abstract states are characterized by the return values of these inspector methods, which are abstracted over to prevent a large number of states. For example, an integer return value is abstracted into one of three abstract values based on its relative value to 0 (either >0, $= 0$, < 0). These approaches may not perform well in the absence of inspectors.

2.1.3 Temporal properties. Several techniques have shown the use of temporal properties in inferring FSA from traces [7, 8, 20, 34, 66]. As mentioned earlier, Lo et al. [41] use temporal properties to prevent erroneous merges. Data mining has been used for the identification of these rules; however, the number of false positives of inferred rules can be high, motivating the need for better ways to identify temporal rules [62]. Le et al. [35] have studied the use of different interestingness measures, while Cao et al. [10] proposed the use of learning-to-rank algorithms composing different interestingness measures to identify accurate properties. Le et al. [34] have also built a meta-model, SpecForge, over existing algorithms in order to decompose mined FSAs into temporal rules, and recompose selected ones back into an FSA. In recent work, Sun et al. [61] used crowdsourcing for identifying correct temporal properties, however, this process was not done automatically, relying on human annotators.

2.2 Test Generation for Specification Mining

Test generation for specification mining have been studied previously. Xie and Notkin [65] propose a feedback loop between specification inference and test generation. However, there is no publicly available version of a tool that implements this strategy and this strategy was not empirically

evaluated. Tautoko [15] uses test generation to further refine a specification. Tautoko mutates an initial test suite and a given FSA model to find missing transitions in the FSA model. DSM [36] uses Randoop [47], which performs randomized test generation, and traces are collected from the test cases generated to train a Recurrent Neural Network on.

The above studies use randomized testing or mutate an existing test suite for mining specifications. These test generation techniques do not systematically diversify the test suite or use any strategy to ensure sufficient diversity in the test cases.

2.3 Search-based test generation

In this study, we use search-based test generation to create test cases to learn specifications from. The generation of test cases are guided towards search goals that we define. We opt to use a search-based test generation tool, Evosuite [25]. Evosuite is a unit test generation tool for Java that uses a evolutionary approach to search for high-quality test cases that fulfils a specified set of coverage criteria. Evosuite evolves a population of tests through multiple generations, and in each generation, discards tests that are less fit while mutating surviving tests. This acts as a search process that iteratively improve the test cases to cover the search objectives. Many optimizations have been proposed and implemented in Evosuite since its inception [5, 58]. Evosuite is automated and does not require any manually written tests as input. Developers can extend the search algorithm or add new coverage and fitness goals. Evosuite comes with a variety of coverage goals, ranging from structural coverage to method coverage goals. Structural coverage goals include line coverage and branch coverage, while method coverage goals guide Evosuite towards tests that invoke every constructor and method of the class. We select Evosuite instead of alternative tools due to its strong performance among state-of-the-art test generation tools [44].

2.3.1 Multiple objective formulation. In the past, test case generation focused on optimising for various coverage criteria independently of each other. Recently, Rojas et al. [57] generated tests while optimising multiple objectives simultaneously, aggregating fitness functions through a weighted sum. However, other studies show the limitations of aggregating multiple fitness goals as a single measure. For example, one such limitation is that the weighted sum aggregation assumes that each fitness goal is independent of each other, which is not true of structural coverage goals (for example, conditional dependencies mean that line and branch coverage goals may depend on one another). Instead of optimizing tests towards a single aggregated fitness value, other researchers have applied multi-objective search algorithms [49]. These algorithms presents several advantages, including preventing the search process from getting stuck in a local minima, and can generate high quality test cases [49]. Indeed, Gay [27] showed that optimising for multiple objectives at the same time instead of enumerating through the objectives one by one lead to test suites that better detect faults.

There are problems specific to test generation when formulated as a multi-objective search. When faced with a large number of search goals, it is impossible to rank many of the individual test cases when considering all of the goals. Due to this, the search process may degrade to become essentially random [49]. Another problem is that a test case that may be fit, when considering every search goal, but may not fully cover any individual search goal. In other words, although the multi-objective formulation of test generation may produce test cases that are pareto-optimal, with the tests representing optimal trade-offs between fitness goals, it may not produce a resulting test suite that completely covers an objective. This is detrimental to our study as we require test cases that contradict a temporal specification, instead of just *being close* to covering it, regardless of the number of other fitness goals the tests are close to covering. Such a set of test cases will provide us with no value. For example, given several test cases which are scored as the vectors shown in

	A	B	C
T1	[0.0	0.9	0.7]
T2	[0.1	0.8	0.7]
T3	[0.3	0.8	0.6]
T4	[0.8	1.0	0.0]
T5	[0.9	0.3	0.9]
	...		

Fig. 1. Example of several objective function vectors of test cases that are non-dominated. There may be more than a few non-dominated test cases and each of them have an equal chance to get included. We use DynaMOSA to address this problem.

Figure 1, all the tests are non-dominated (each test is no worse than another with respect to at least one search goal). In this example, the individual values in the vector represent the distance for a particular search goal. A lower distance is better and a distance of 0.0 indicate that the test case covers that goal. While objective A is covered by test T1 and objective C by T4, objective B is not covered by any test. As none of the tests dominate each other, they have equal probability of getting selected for the next generation. Objective B is a difficult objective to cover. In our study, it is important that we retain and evolve test case T5 in the next generation as it is closest to covering objective B.

To address these problems, the DynaMOSA multi-objective algorithm [50] has been proposed for Evosuite. The DynaMOSA algorithm, at a high-level, is given in Algorithm 1. It evolves an initial randomly generated population of tests through multiple generations. Given a set of coverage goals, the algorithm evolves a population of test cases through the usual mutation and cross-over operators (line 5). This gives us the offspring, Q, a set of test cases containing new tests as well as retaining some test cases from the previous generation. The test cases in Q are ranked and binned into a list of fronts, which partitions Q. The first front contains the best test case with respect to each coverage goal. After the first front, each subsequent front ranks the remaining test cases by their pareto-optimality. The length (number of statements) of the test case is used as a tiebreaker when two test cases have the same score, preferring shorter test cases which is more likely to run in shorter time. Then, the top ranked test cases form the population of the next generation and the offspring of this population are generated, and the process continues until the search budget is exhausted.

2.3.2 Archive. DynaMOSA maintains an archive, A (used in lines 8 and 23), similar to other search-based test generation strategies. During the test generation process, the archive stores test cases covering previously uncovered goals and provides a way to retrieve the best test case for a particular search goal. The archive accounts for accidental coverage; a goal may be collaterally covered by a previous search for another set of goals. When a test case for a particular search goal is stored in the archive, this search goal is removed from the current set of goals (line 8). As a consequence, the current set of search goals contains only the uncovered goals and focuses the search process on them. The archive is updated whenever the search goal is covered, but also when a test case that is shorter than the current test and covers the same goal. At the end of the test generation process, the test cases in the archive are retrieved and are the output test suite (line 22).

2.3.3 Dynamic selection of targets. The key feature of DynaMOSA is that it allows Evosuite to dynamically select targets based on the control dependencies between one another. Initially, only a

Input: A set of coverage goals C.
Input: Program, P.
Input: Population Size M.
Output: A test suite, TS, which is a collection of test cases

```
1  D = GetControlDependencies(P)
2  P = RandomPopulation();
3  A = InitArchive(P, C);
4  C' = UpdateCurrentGoals(A, C, D);
5  while Search budget is not expanded do
6  |  Q = GenerateOffspring(P);
7  |  P = {};
8  |  A = UpdateArchive(A, Q, C');
9  |  C' = UpdateCurrentGoals(A, C', D);
10 |  fronts = Rank(Q);
11 |  for front ← fronts do
12 |  |  if P.size >= M then
13 |  |  |  break;
14 |  |  end
15 |  |  for TC ← fronts do
16 |  |  |  if P.size + 1 > M then
17 |  |  |  |  break;
18 |  |  |  end
19 |  |  |  AddToPopulation(P, TC);
20 |  |  end
21 |  end
22 end
23 TS = A.getTestCases();
```

Algorithm 1: Simplified version of the DynaMOSA algorithm. Given a program, P, and the set of coverage goals, C, DynaMOSA constructs a test suite, TS.

```
1  if (functionA())) {  // Initially targeted as it does not depend on another line
2    if (functionB()) { // Targeted only after line 1 is covered
3      functionC();     // Targeted only after lines 1 and 2 are covered
4    }
5  } else {
6    functionD();       // Targeted only after line 1 is covered
7  }
```

Fig. 2. DynaMOSA uses the control dependencies to dynamically target only search goals that can be covered. Line 3 is targeted only after lines 1 and 2 are covered. After lines 1 and 2 are covered, line 3 is added to the current set of goals.

subset of the coverage goals that are independent of other goals are targeted. Dynamically selecting targets allows Evosuite to be more efficient when trying to cover multiple structural goals.

For example, statements within branches require the if-statement to be covered first, therefore these statements are initially not targeted by Evosuite until the if-statement has been covered. If the if-statement has not been covered, then these goals cannot be covered. The fitness, of a test

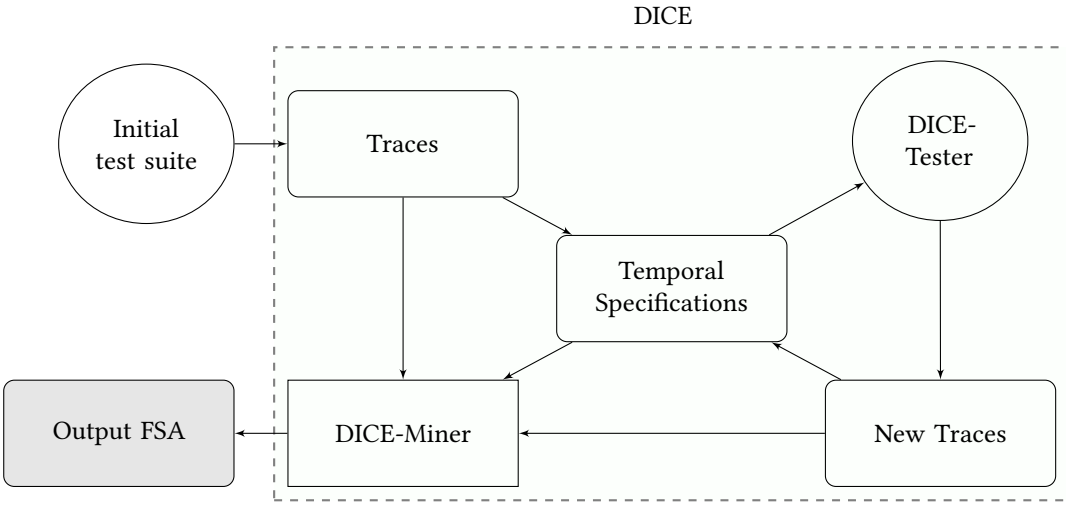

Fig. 3. High-level overview of DICE

with respect to these goals is, therefore, always worse than the fitness of the test with respect to goal of covering the if-statement. In the example shown in Figure 2 statement 3 cannot be covered before both statements 1 and 2 are covered. If a test case has not covered statement 1, it is not necessary to consider the fitness value of a test with respect to the search goal of covering statement 3. Therefore, these uncoverable goals do not contribute meaningfully to the ranking.

Evosuite first computes a control dependency graph of the program before generating tests and uses information from the control dependency graph to update the current set of search goals. As described earlier, before ranking test cases by their pareto-optimality, DynaMOSA first ensures that tests that are closest to a targeted search goal always survive and are retained in the next generation, even if these tests are not pareto-optimal with respect to the other goals. This makes it more probable for Evosuite to progress towards individual goals, including those that may be difficult to cover. This particular feature is the reason why we build DICE on top of the DynaMOSA strategy.

In Algorithm 1, the dynamic selection of targets can be seen in lines 4 and 9, in which the archive is used to determine which goals have been covered. DynaMOSA uses the control dependencies, D, to determine the initial set of targets in line 4. As the goals are covered, it adds the goals that depend on the covered goals in line 8 to the currently targeted set of goals, C'.

3 DICE APPROACH

3.1 Overview

We show a high-level overview of the approach used by DICE in Figure 3. DICE consists of 2 main components: DICE-Tester and DICE-Miner. From a high-level perspective, DICE takes a class under test and an initial test suite as input, producing a FSA model as output. DICE first exercises the test suite, collecting the execution traces. This is followed by three phases:

- **Mining Purity-Aware Temporal Specification.** First, temporal specifications, in the form of LTL temporal properties, are mined from these traces while being aware of method purity.

- **Adversarial Test Generation.** The temporal specifications are fed into DICE-Tester and are converted into search goals for the test generation process. DICE-Tester is adversarial to the temporal specifications and refines them by invalidating incorrect properties, while generating new test cases and collecting the execution traces of these test cases.
- **FSA Inference.** Finally, the new traces and temporal specifications, with invalid specifications now removed, are input into DICE-Miner, which will infer an FSA. DICE-Miner avoids weaknesses of existing algorithms by using method purity and the temporal specifications to prevent over-generalisation.

3.2 Mining Purity-Aware Temporal Specification

The test generation phase of the adversarial specification mining process requires a set of specifications. As such, the first phase of DICE is to mine temporal specifications over the input traces collected from the input test suite. A formalism over constraints ordering events is Linear Temporal Logic (LTL) [32, 55]. Like previous studies in specification mining [7, 8, 20, 34, 66], we use LTL to specify constraints over events. In this work, events are specifically method invocations of a class and the following subset of LTL connectives are used in the property templates:

(1) X ϕ means that ϕ has to hold at the ne**X**t state.
(2) F ϕ means that ϕ has to hold at some **F**uture state.
(3) G ϕ means that ϕ has to hold **G**lobally at all future states.
(4) ρ U ϕ is '**U**ntil', which means that ϕ has to hold at some point. ρ has to hold until ϕ holds.
(5) ρ W ϕ is '**W**eak until', which means that ρ has to hold until ϕ holds. If ϕ never becomes true, ρ has to hold forever.

Six LTL property templates are commonly used in previous studies. The six LTL property templates are described as follows:

(1) AF(a, b): an occurrence of event a must be eventually followed by event b. In LTL, this rule is $G(a \rightarrow XFb)$
(2) NF(a, b): an occurrence of event a is never followed by event b. In LTL, this rule is $G(a \rightarrow XG(\neg b))$
(3) AP(a, b): an occurrence of event a must be preceded by event b. In LTL, this rule is $\neg aWb$
(4) AIF(a, b): an occurrence of event a be immediately followed event b. In LTL, this rule is $G(a \rightarrow b)$
(5) NIF(a, b): an occurrence of event a is never immediately followed by event b. In LTL, this rule is $G(a \rightarrow X(\neg b))$
(6) AIP(a, b): an occurrence of event a must be immediately preceded by event b. In LTL, this rule is $F(a) \rightarrow (\neg a\ U(b \wedge Xa))$

The last three properties, introduced by Beschastnikh et al. [7] and Le et al. [34], are "immediately" variants of the first three properties and have been shown to be useful for describing FSAs. In this work, we use the LTL property templates from previous work which only consider 2 events. Later, in Section 5.4 (Qualitative Evaluation), we note some limitations of considering only 2 events at a time. We also note that the primary objective of this study is to infer automata models, and the LTL specifications are only later used to guide the testing process and the inference of the models.

While we use the same LTL property templates studied by Beschastnikh et al. [7] and Le et al. [34], in our work, we adapt three of them. Recently, Sun et al. [61] pointed out shortcomings of these patterns when using crowdsourcing to identify temporal specifications. AIP(a, b) and AIF(a, b), for example, can never be true since a method can always be invoked between any pair of events a and b. To address these shortcoming and to retain the benefits of these properties in describing temporal constraints, we observe that side-effect-free method invocations can never

affect the state of a software system, and as such, can be abstracted away in the description of the "immediately" variants of the LTL properties. We reformulate these variants to incorporate knowledge of side-effect free methods:

(1) AIF(a, b): an occurrence of event a must be immediately followed by an occurrence of event b , ignoring all occurrence of side-effect-free events. In LTL, this rule is $G(a \rightarrow X((p_1 \vee p_2 \vee p_3 \vee \cdots \vee p_n)\, U\, b)$, where $p_1, p_2, \ldots, p_n$ are side-effect free events, known ahead of time.
(2) NIF(a, b): an occurrence of event a is never immediately followed by event b, ignoring the occurrence of side-effect-free events. In LTL, this rule is $G(a \rightarrow X(p_1 \vee p_2 \vee p_3 \vee \cdots \vee p_n)\, U\, \neg b)$, where $p_1, p_2, \ldots, p_n$ are side-effect free events, known ahead of time.
(3) AIP(a, b): an occurrence of event a must be immediately preceded by event b, ignoring the occurrence of side-effect-free events. In LTL, this rule is $F(a) \rightarrow (\neg a\, U(b \wedge X((p_1 \vee p_2 \vee p_3 \vee \cdots \vee p_n)\, U a)))$ where $p_1, p_2, \ldots, p_n$ are side-effect free events, known ahead of time.

To identify side-effect free methods of a class, we use a lightweight static analysis [30] and a heuristic based on the method name (we consider names starting with "is-" or "has-" to be getters, which are typically pure). While static analysis is used to partially accomplish this, it is not necessary for our approach. When neither source code nor bytecode is available, a developer can annotate the purity of relevant events by hand.

To mine LTL specifications, we use a solution similar to the linear miner algorithm described by Lemieux et al. [38]. However, as we are only interested in 2-event rules and we restrict ourselves to a few properties, we do not face the challenges that they solve. We use a simple way of iterating over the traces and try to use each trace to falsify the LTL specifications.

In the work by Lemieux et al. [38], support and confidence thresholds are needed. The support counts the number of times while iterating through the traces where a property can be falsified, but is not. The confidence of a property is the ratio of support of a property to the number of times the property can be falsified. When the number of times the property can be falsified is 0, the confidence is defined to be 1.

For our work, as we are interested only in properties that are never falsified, thus in our implementation, we require a confidence of 1.0 for the rules that we mine. Moreover, we only require a support of 1 to admit the temporal property. In other words, as long as a property holds on a trace, and we do not encounter any trace that contradicts the property, we admit the property.

3.3 Adversarial Test Generation

The specification mining process adversarially generates test cases against an input specification, aiming to invalidate the specification. To this end, DICE converts the temporal specifications mined from the previous phase into search goals for search-based testing. The objective of this phase is to allow for the discovery of test cases that produce traces that are uncommon and not represented in the initial test suite. To do so, the temporal specifications are converted into fitness goals for test generation. DICE-Tester generates test cases using Evosuite[1] with the addition of the new fitness goals and coverage criteria.

A fitness cost is computed for each fitness goal for each test case that is generated. A single test case may contain multiple object instances of the class of interest and we consider a single trace to be the methods invoked on a single object instance. Therefore, each test may produce multiple traces, one for each object instance in the test case. The fitness of a test, T, with respect to a fitness goal, G, is determined by the trace with the best fitness.

[1]Evosuite version 1.0.6 was used in this study

$$Fitness(T, G) = min(Fitness(tr, G)) \ , \ tr \in traces(T)$$

We define a new coverage criterion, `TemporalPropertyCounterExample(PropertyType, EventA, EventB)`, based on the LTL properties we have mined. An temporal property is covered if a trace contains a sequence of method invocations that is a counterexample of it. In this formulation, DICE-Tester creates a fitness goal for each temporal property we have mined, guiding Evosuite to produce counterexamples for them. With respect to a single fitness goal, a test is fitter than another if the fitness cost of the test is lower than the other. A goal is covered when the fitness cost is 0, i.e., when a counterexample trace is produced when exercising the test.

When the property is not covered, we aim to guide the test generation process towards a test covering the property. Generally, a trace is scored relative to the number of modifications required to transform the trace to, first, a trace supporting the temporal property, then, to falsifying the temporal property. This will push test generation towards test cases that first support the property, from which they may be mutated towards counterexamples afterwards.

To this end, we grant a better fitness cost when the trace contains one method of a "never" property (NF, NIF) and when the trace contains both methods of an "always" property (AP, AIP, AF, AIF). To summarize, for a temporal property AP(A,B) or AIP(A,B), we assign fitness costs to traces such that the following ordering holds:

(1) Traces falsifying the temporal property (best)
(2) Traces supporting the temporal property
(3) Traces containing at least one of the events, A or B, but neither supporting or falsifying the temporal property.
(4) Traces where none of A and B are present (worst)

The above ordering guides our design of the fitness functions for all property types. Using the above ordering, the traces are partitioned into 4 subsets for the "always" properties, AP, AIP, AF, and AIF. In our implementation, as Evosuite expects a numerical cost, we evenly partition the range [0, 1] into the 4 subsets. Formally, each trace, tr, is assigned a cost for the search goal targeting a property, p, relating two events, A and B, based on the following conditions:

$$Fitness(tr, p) = \begin{cases} 0.0 & \text{if falsifies(tr, p)} \\ 0.33 & \text{if !falsifies(tr, p) and supports(tr, p)} \\ 0.66 & \text{if !falsifies(tr, p) and !supports(tr, p)} \\ & \quad \text{and (contains(tr, A) or contains(tr, B))} \\ 1.0 & \text{otherwise} \end{cases} .$$

To provide some intuition, we use an example using a property AP(A, B), event B must precede event A. A trace in which event B does not precede event A, e.g. [X, Y, A], will falsify the property. Given a trace [B, X, Y, A], which supports the property, the trace is one transformation away, in which the method call B is removed, away from falsifying the property. For another example, using a trace [B, X, Y]. This trace is one edit from supporting the property. The event A is added in any of the three positions (between B and X, between X and Y, after Y). After this, the modified trace is one edit from falsifying the property AP(A, B). As it took two edits, the trace [B, X, Y] is less fit than the trace [B, X, Y, A] which only requires one edit. Observe that a trace may containing sequence of events that both supports and falsifies a property. A trace [A, B, A] has both the sequence [A], which falsifies AP(A,B), and [B, A], which supports AP(A,B). According to our fitness formulation, this trace will obtain a fitness cost of 0, since it falsifies the property.

A trace without both events A and B, e.g. [X, Y], has the worst fitness cost for the AP(A, B) search goal. This may be contrary to intuition, since a trace without any of the property's events may resemble a trace that has a few edits away from falsifying the property. For example, the trace [X, Y] is only 1 edit from falsifying the property directly, e.g. [X, Y] to [X, Y, A]. There are numerous such traces among the entire test population. An objective of our fitness functions is to focus the search on properties that are hard to falsify. If the property can be easily covered by having such a trace transformed to a counterexample trace, there would be numerous such cases. It is very likely that the search goal can be covered as part of collateral coverage when the entire test population evolves to satisfy other coverage goals. On the other hand, temporal properties that are more difficult to cover will benefit from the focused search.

For example, consider the property AP(`isEmpty:false`, `push:true`) for a hypothetical data structure, which holds if the data structure cannot report that it is not empty unless the push method has been successfully invoked. A counterexample trace can be constructed, e.g. [`<init>`, `pushAll:true`, `isEmpty:false`], by using an alternative method (`pushAll:true`) to insert an item into the data structure. From a trace supporting the event, e.g. [`<init>`, `push:true`, `isEmpty:false`], a single transformation from `push:true` to `pushAll:true` will lead to a trace falsifying the property. On the other hand, given an initial trace [`<init>`, `get:false`, `clearAllElements`] that does not contain either event in the property, if we try to falsify the property by adding a `isEmpty` method invocation, instead of adding the `isEmpty:false` event, the event `isEmpty:true` added to the trace. It is not possible to arbitrarily add an `isEmpty:false` event into any position in a trace. The `pushAll:true` event has to be successfully added first. As a result, a randomly generated trace without the `isEmpty:true` event, e.g. [`<init>`, `get:false`, `clearAllElements`], is much further than a trace supporting the property, e.g. [`<init>`, `push:true`, `isEmpty:false`].

For a temporal property AF(A,B), a trace is scored similarly. Observe that the fitness cost of AF(A,B) can be computed by using the same scoring rules for AP(B, A) and reversing the trace. Likewise, AIF(A,B) can be computed using AIP(B, A) by reversing the trace.

For a NF property, NF(A,B), the same ordering of traces described above applies. A trace supporting the property is a trace where A is present, but B does not appear after A. In the "immediately" variation of NF, NIF, the fitness functions can be computed to differentiate individual traces from one another with higher granularity, while still respecting the ordering of traces. In NIF, the fitness cost reflects how closely positioned the two events in the property are within the trace. A better fitness cost is returned if both events are included in the trace, and we score the fitness value based on how far apart the method invocations are. This will push test generation towards tests where the events are located nearer to each other. The fitness cost of a trace, tr, with respect to the fitness goal of the property NIF(A,B) is computed as shown in Algorithm 2.

Algorithm 2 takes a single trace, consisting of a sequence of events, as input and returns the fitness score of this trace. The fitness score ranges from 0 to 1, and a score of 0 indicates the maximum fitness value while a score of 1 indicates that the trace has no relevance for this goal. A single pass is made through the sequence of events to look for instances of event A (lines 5-19). A counter (line 4) to measure the distance between a pair of events A and B is updated in this pass. Each time we reach an instance of event A, we record its position and reset the counter (lines 7,8). Once an instance of event A has been found (line 10), we increase the counter for each impure (i.e. not side-effect free) event we pass (line 11-12). Whenever we reach an instance of event B, we update the minimum distance between the last seen instance of A and B if the counter is less than the previous minimum distance between A and B (line 14-16). Finally, the fitness score is computed from the minimum distance. If the distance is 0, then the goal is covered and 0 is output as the fitness score (line 20-22). Otherwise, the ratio of the distance to the length of trace is used as the fitness score (line 23-27).

Input: A trace, *tr*. A fitness goal for a NIF property, $NIF(A, B)$.
Output: Fitness cost of the trace. 0 means that the property is covered

```
1  eventAPosition = -1;
2  i = 0;
3  distance = 999;
4  counter = 0;
5  for event ← tr do
6      if event = A then
7          eventAPosition = i;
8          counter = 0;
9      else
10         if eventAPosition ≠ −1 then
11             if !isPure(event) then
12                 counter += 1;
13             end
14             if event = B then
15                 distance = Min(counter, distance);
16             end
17         end
18     end
19 end
20 if distance = 0 then
21     return 0
22 else
23     if eventAPosition = -1 then
           /* Event A does not appear, trace is irrelevant to this goal      */
24         return 1
25     else
26         return distance / (length(tr))
27     end
28 end
```

Algorithm 2: Fitness computation of a trace, tr, with respect to `TemporalPropertyCounterExample(NIF, A, B)`

To conclude, the satisfaction criterion of each *TemporalPropertyCounterExample* depends on the property type. Effectively, if a test contains a trace with a fitness score of 0, then the test is a counterexample. It may be interpreted as follows:

- AF(a,b): a test is a counterexample of AF(a, b) if it contains a trace with an invocation of a that is not (immediately) followed by b.
- NF(a,b): a test is a counterexample of NF(a, b) if it contains a trace with an invocation of a that is (immediately) followed by b.
- AP(a,b): a test is a counterexample of AP(a, b) if it contains a trace with an invocation of b that is not (immediately) preceded by a.

As a consequence of including our new fitness goals into Evosuite, there are a large number of search objectives (one for each temporal property we have mined). As described in Section 2,

this hampers the ability of Evosuite to search effectively for good test cases that will cover the uncovered goals. The large number of search goals causes the search process to be similar to a random search process, reducing its performance. To manage the large set of goals, we use the DynaMOSA search algorithm as discussed earlier in Section 2. For each search goal, the DynaMOSA algorithm uses a test ranking strategy which ensures that the test case closest to covering it always survives to the next generation when tests are ranked and weaker tests are discarded. Furthermore, it models the dependencies between structural goals. At any time, only the goals with fully satisfied dependencies are in the current set of goals. This strategy prevents goals that cannot contribute meaningfully to the test ranking from becoming targets. However, we found that the existing implementation of the DynaMOSA search algorithm was not sufficient to enable Evosuite to find counterexamples effectively.

Thus, we use the DynaMOSA search algorithm in DICE-Tester with three modifications. To further boost the efficiency of selecting tests, our first modification is to model the dependencies between the coverage goals beyond structural goals to allow Evosuite to dynamically select goals more effectively. While the DynaMOSA algorithm already models the dependency between structural coverage goals based on control dependencies, it does not model other forms of dependencies, including usage dependencies between methods. It is unable to perform any reasoning about method coverage goals, which DICE-Tester is able to constrain using the LTL properties we have mined previously.

The dependency tree is constructed prior to the start of the search process. At this stage, we assume that every mined LTL properties is true. This dependency tree relates method coverage goals with the `TemporalPropertyCounterExample` coverage goals that we introduced in this work. A method coverage goal, `Method(A)`, is covered when a test case invokes the method A at least once. We add dependencies based on the following rules:

- Given AP(A, B), DICE-Tester adds a dependency where B depends on A. i.e. `Method(A)` should be covered before `Method(B)`
- Given NF(A, B) or NIF(A,B), DICE-Tester adds a dependency where `Method(A)` should be covered before `TemporalPropertyCounterExample(NF, A, B)` or `TemporalPropertyCounterExample(NIF, A, B)`.
- Given AF(A, B) or AIF(A,B), DICE-Tester adds a dependency where `Method(A)`should be covered before `TemporalPropertyCounterExample(AF, A, B)` or `TemporalPropertyCounterExample(AIF, A, B)`.
- Given AP(A, B) or AIP(A,B), DICE-Tester adds a dependency where `Method(B)` should be covered before `TemporalPropertyCounterExample(AP, A, B)` or `TemporalPropertyCounterExample(AIP, A, B)`.

For example, given `NF(StringTokenizer, nextToken)`, for its corresponding goal `TemporalPropertyCounterExample(NF, StringTokenizer, nextToken)` to be satisfied, the method coverage goal, `Method(StringTokenizer)`, must be satisfied first. If this method coverage goal is not satisfied, then it is impossible for the test to produce a counterexample and this goal is always completely uncovered for any test case. Thus, it will not contribute meaningfully to the ranking of tests maintained by DynaMOSA [25]. Constraints about the ordering of methods provided by the set of AP temporal properties also allow us to add dependencies between a pair of method coverage goals. For another example, given `AP(initVerify,update)`, we add a dependency between `Method(update)` and `Method(initVerify)`, requiring an invocation of *initVerify* before Evosuite adds *update* to the set of currently targeted goals. While other coverage goals may return a fitness value between 0 and 1, the fitness of a test with respect to a method coverage goal is

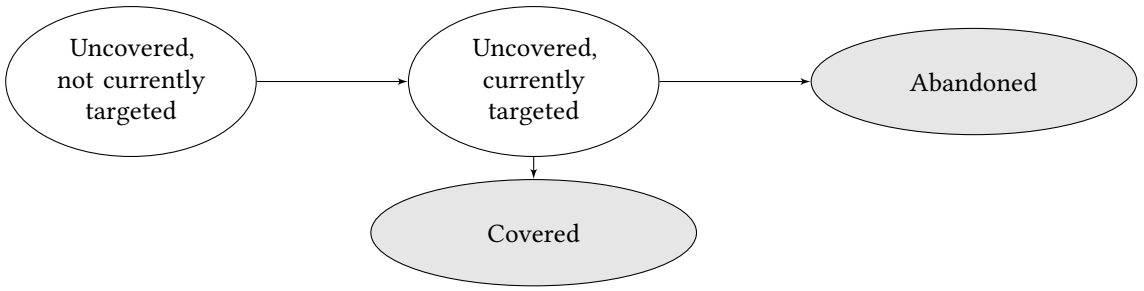

Fig. 4. Lifecycle of a search goal. At the end of the search process, a goal is either covered or abandoned.

binary, either the goal is covered or it is not. Therefore, if the prerequisites are not met, a method coverage goal is always uncovered for every test case.

Our second modification is made for the lifecycle of a search goal. In the DynaMOSA algorithm, goals are in one of 3 states. A goal is either a) covered, b) uncovered but not in the current set of goals, or is c) uncovered but in the current set of goals. In DICE-Tester, we added a new state in this lifecycle. Thus, a goal can be in one of 4 states: covered, uncovered but in the current set of goals, uncovered but not in the current set of goals, or *abandoned* (see Figure 4). We refer to a goal as *abandoned* if it is not covered but we have removed it from the current set of goals.

The addition of this state is motivated by the fact that at least some of the mined properties are true. As these properties are true, it is impossible to generate a test case that is a counterexample to it. Consequently, goals representing counterexamples to these properties can never be covered, and this has detrimental effects on the search process. Such goals will continue to contribute to the preference ranking used in Evosuite. This has several implications. Test cases that are closest to and "almost covering" the goal will always remain in our population, although they will not contribute to producing interesting test cases. The search may be weighted towards tests that have a higher chance of covering these goals, even though these goals cannot be covered. This may potentially prevent other test cases (that may lead to a counterexample of another property) from getting added to the population. Furthermore, these goals have no meaningful contribution to the ranking process when weighing the pareto-optimality of the other test cases. This slows down the search process, wasting time to compute the fitness of tests with respect to these goals.

Hence, we add the abandoned state in the lifecycle of a goal. To enable goals to transit to this state, DICE-Tester tracks the age of each goal. Covered goals and goals that are not in the set of currently targeted goals do not have an age. When an uncovered goal moves into the set of targeted goals, its age is initialized to 0. We increment the age of a goal for each generation where DICE-Tester does not find a test case that covers it. Once the age of a goal has exceeded a threshold, we abandon the goal by removing it from the set of targets. In our experiments, we set this threshold to 100 generations. Once abandoned, goals can never be restored back to the set of targets.

Our third modification is to reset the population of the tests once it has gotten stuck. While the first two modifications allow Evosuite to find more counterexamples, we observed that the search can still get stuck in a local optima. Finally, we bypass this problem by resetting the population of tests, and in effect restarting the search, once 100 generations has passed without DICE-Tester finding a test that is a counterexample to *any* goal. We did not thoroughly empirically evaluate the threshold for abandoning a goal or resetting the population, however, we noticed in our experiments that changing these parameters do not affect the results much, provided that they are not too small.

With these three modifications, DICE-Tester is able to guide test generation towards finding counterexamples to spuriously mined LTL properties, to invalidate them. Temporal properties with at least one counterexample are removed.

Next, we collect execution traces from the output test suite, which are constructed from the test cases that produced counterexamples to the temporal properties. These are later passed as input to the FSA inference process. Typically, specification mining algorithms use traces of correct executions and our approach is not an exception. Therefore, we filter out traces of executions that may not represent a correct usage. If the invocation of a method results in a thrown exception, we ignore the invocation and any further method invocations (as the exceptional invocation may have an effect on the state). Apart from exceptional executions, we also try to detect resource leaks and omit any possible trace that caused them. We keep track of the number of file descriptors that are opened by the process that is executing the tests [2]. Next, we compared the number of file descriptors before and after the runs, and if we find a mismatch between them, we assume that the tests have triggered a sequence of methods that leaked a file or a socket. When this happens, we remove all traces of tests that led to resource leaks. While this may inadvertently result in correct traces that are incorrectly discarded (consider a scenario where a test instantiated multiple FileOutputStreams, we discard traces from every FileOutputStream as long as one of them caused a leak, even if the rest of the instances represent correct usage of FileOutputStream), this helps us ensure that the traces we have collected do not contain executions of invalid usage of an API or class.

3.4 Example of the search process

```
interface DataStructure<T> {

    public boolean add(T item);

    public boolean addAll(Collection<T> items);

    public boolean isEmpty();

    public void clear();

    public T get();

    public Collection<T> getAll();
}
```

Fig. 5. The API of a hypothetical data-structure.

In this section, we present a synthetic example of the search process. Using a hypothetical data-structure shown in Figure 5, we show how a small set of test cases may evolve over a few generations to become counterexamples for a three properties that are not true properties:

- Goal 1: AP(isEmpty:FALSE, add:true),
- Goal 2: AIF(clear, isEmpty:TRUE), and
- Goal 3: AF(clear, isEmpty:TRUE),

We run the DICE process to search for test cases that will falsify the properties. In this simple example, all three properties are falsifiable. As described earlier, we use the dependencies between search goals to consider fewer search goals at a time, omitting goals that cannot be covered yet. At

[2]We run Evosuite without its sandbox which prevents environmental interactions

the start of the DICE-Tester process, the following dependencies between search goals are added:

$Method(isEmpty : FALSE) \rightarrow AP(isEmpty : FALSE, add : true)$
$Method(isEmpty : TRUE) \rightarrow AIF(clear, isEmpty : TRUE), AF(clear, isEmpty : TRUE)$
$Method(clear) \rightarrow AIF(clear, isEmpty : TRUE), AF(clear, isEmpty : TRUE)$
$Method(add : TRUE) \rightarrow AP(isEmpty : FALSE, add : true)$

The dependencies should be interpreted such that the search goals on the right (the consequent) are added to the currently targeted set of goals once the search goal on the left (the antecedent) has been covered. The test generation process starts by creating a population of randomly generated tests. In this example, we assume that the population size is two, and that each test produces only one trace. First, DICE-Tester generates 2 tests, giving us the following traces as shown below in Figure 6:

```
[clear, isEmpty:TRUE, getAll]                    (-, 0.33, 0.33)
[getAll, add:FALSE]                              (-, 1.00, 1.00)
```

Fig. 6. The initial population. The numbers on the right show their fitness costs for Goals 2 and 3. Goal 1 is not considered yet as neither of the method coverage goals it depends on has been covered.

In Figure 6, Goal 1 is not considered among the search goals (listed as '-') as neither of the method coverage goals it depends on has been covered. When the number of goals is large, this helps to simplify the comparison between test cases. When this initial population of test cases is evolved, it produces a set of new test cases as shown in Figure 7. Since the method coverage goal of Goal 1 has been covered, all the search goals for the 3 properties are now considered.

```
[clear, getAll, isEmpty:TRUE, getAll]            (1.00, 0.00, 0.33)   *
[clear, isEmpty:TRUE, add:TRUE, isEmpty:FALSE]   (0.33, 0.33, 0.33)   *
[add:TRUE, clear, isEmpty:TRUE, getAll]          (0.66, 0.33, 0.33)
[getAll, isEmpty:TRUE, add:TRUE, getAll]         (0.66, 0.66, 0.66)
[getAll, add:TRUE, get]                          (0.66, 1.00, 1.00)
```

Fig. 7. The offspring of the first generation. We order them by their pareto-optimality.

In Figure 7, the test cases that are selected to form the next population of tests are indicated with a *. As the first test case has fully covered Goal 2 (falsifying the property that `clear` is always followed by `isEmpty:TRUE`), it is selected. As for the second test case, it ties with the third test case for the best fitness score on Goal 3, but it is the best test case for Goal 1. Therefore, both the first and second test cases are selected as they are the best test cases with respect to the search goals. The remaining test cases are ordered by pareto-optimality. The third test case dominates the fourth test case, and the fourth test case dominates the fifth test case. The first test case is inserted into the archive, as it has fully covered a search goal. If the population size was greater than 2, then the remaining test cases would be added in this order.

Finally, in the next round, we get the following offspring shown in Figure 8. As search goal 2 was already fully covered earlier, we no longer need to consider the test cases' fitness with respect to it.

Both Goals 1 and 3 have counterexample traces from this set of offspring, and these two test cases are added to the archive. As all the search goals related to the temporal properties are covered, we end the test generation process. In practice, it is typically the case that not all coverage goals can be covered and the test generation process only ends when the search budget is fully consumed.

```
[clear, isEmpty:TRUE, addAll:TRUE, isEmpty:FALSE]        (0.00, -, 0.33)   *
[clear, clear, getAll]                                   (0.66, -, 0.00)   *
[clear, getAll, isEmpty:TRUE, add:TRUE, getAll ]         (0.66, -, 0.66)
[clear, isEmpty:TRUE, sEmpty:TRUE, add:TRUE, getAll]     (0.66, -, 1.00)
```

Fig. 8. The offspring of the second generation of test cases. All the test cases are covered after this. Note that Goal 2 was not considered when computing the objective function vectors as it has a corresponding test case in the archive.

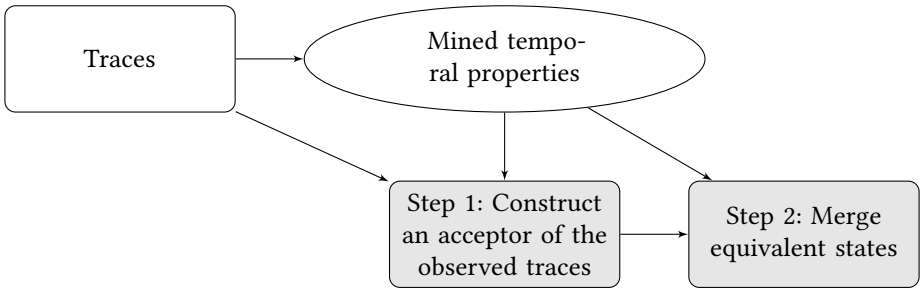

Fig. 9. High-level overview of DICE-Miner

The test cases from the archive are retrieved to form the output test suite, which is run and its execution traces are collected to be used as input to the next step of DICE.

3.5 FSA Inference

The final phase of the adversarial specification mining approach is to take the temporal properties and traces produced from the previous phases and infer an FSA model. In DICE, the DICE-Miner algorithm is responsible for this. We add the traces produced by the DICE-Tester to the original set of traces for input to DICE-Miner.

An overview of DICE-Miner is given in Figure 9. The DICE-Miner algorithm to infer an FSA-based specification is similar to the k-tails algorithm, comprising of two steps. In the first step, we construct an initial automaton based on the input traces and the mined temporal properties. Similar to the PTA (described earlier in Section 2), our initial automaton accepts all of the observed input traces. The mined temporal properties are used to prevent erroneous merging of states. In the second step, we merge equivalent states in this automaton, leveraging the mined temporal properties. The mined temporal properties are used to derive the set of enabled methods of each state, which is the equivalence criteria used in DICE-Miner to merge states. We elaborate on this equivalence criteria below in Section 3.5.2.

Regarding the first step of the algorithm, we make the following observations:

- Observation A: Side-effect-free events can be interleaved in any order, and do not change the present state of the software system.
- Observation B: The first step of constructing a PTA makes the assumption that states with the same history of events are equivalent. This assumption may not be true.

We propose a specification mining algorithm with these observations in mind. In light of the first observation, we include information of the purity of each method to allow the model to have greater generalizability. The second observation guided us to prevent the incorrect conflation of states while constructing an initial model that accepts all of the input traces.

With observation A, we model the freedom of side-effects as self-loops in the automaton. We add Constraint A where we ensure that transitions labelled with side-effect free method calls are self-loop. This has the advantage of increasing the model's generalizability, allowing it to accept an equivalent trace with a different permutation of the pure methods. For example, for `StringTokenizer`, observing an input trace [`StringTokenizer`, **`hasMoreElements:true`**, `hasMoreTokens:true`] allows the construction of an automaton that will accept a different trace (note a different ordering of the method invocations), [`StringTokenizer`, `hasMoreTokens:true`, **`hasMoreElements:true`**]. This is the case even without the observation of a trace with this sequence of method calls, since the only difference is that the pure methods *hasMoreTokens* and *hasMoreElements* were invoked in a different order from the same state.

To address observation B, we add Constraint B where we prevent the erroneous conflation of states occurring in the first step of k-tails and its variants. If these states are incorrectly merged in the first step, regardless of the equivalence criteria selected for merging states, this inaccuracy in the initial model will negatively impact the quality of the final FSA produced. This is because the second step does not split up incorrectly merged states from the first step. For an example of Observation B, with the `Iterator` class, given the trace [**`Iterator`, `hasNext:true`, `next`**, `hasNext:false`] and a different trace sharing a prefix, [**`Iterator`, `hasNext:true`, `next`**, `hasNext:true`], the states reached in the two traces after the invocation of **next** is different, yet constructing a PTA will conflate these states as they share the same preceding events [`Iterator`, `hasNext:true`, `next`]. Conflating these states produces an automaton where both `hasNext:false` and `hasNext:true` are incorrectly enabled from the same state.

To address this, we propose to detect incorrectly merged states using the mined temporal properties while constructing an initial automaton accepting all of the observed traces. Whenever adding a new transition to an automaton results in a automaton that may produce traces violating the mined properties, we modify the automaton such that these violations will not occur. Next, we discuss the details of the DICE-Miner algorithm, and show we address both observations within the first step of our algorithm.

3.5.1 First Step. Next, we describe the first step of the DICE-Miner algorithm. In the first step, we pass the example traces into the function *CreateCompatibleAcceptor*, which constructs a Nondeterministic Finite Automaton (NFA). This automaton is build to accept all of the example traces, much like a PTA. However, unlike the construction of a PTA, *CreateCompatibleAcceptor* avoids the creation of states that may accept sequences of events that violates a constraint (we refer to such states as *incompatible* with the constraint). We consider a state to violate a constraint if it causes the automaton to accept a trace violating the constraint. Observe that although the states were constructed based on individually observed traces, and that every trace does not violate the temporal properties mined earlier, it is possible to construct a PTA with states that may accept traces violating the temporal properties. For example, given two traces, [`Stack`, `addAll`, `remove`, `isEmpty:TRUE`] and [`Stack`, `addAll`, `remove`, `get`] and a temporal property, *NIF(isEmpty=True, get))*, a state with an incoming edge, `isEmpty:TRUE` (a self-loop on the state, as a result of Constraint A and the fact that `isEmpty` is pure), and an outgoing edge, `get`, is *incompatible* with the temporal property as it can accept a trace with the sequence of events [`isEmpty:TRUE`, `get`]. The algorithm to split up a state with incompatible edges is given in Algorithm 3.

CreateCompatibleAcceptor first initializes an empty automaton before iterating over each trace (lines 1-2). For each event in a trace, we first try to accept the event without modifying the NFA (lines 5-8). On reaching an event, *e*, that cannot be accepted, we modify the NFA to add new states and transitions such that it will accept the events. Before adding a new state, we first ensure that the new transition will not cause the current state to be incompatible with any constraint (line

```
Input: Input traces, traces
Output: A DFA that accepts all traces in traces, without creating any states that may violate
        any constraint
1  automaton = emptyAutomaton();
2  for trace ← traces do
3      prefix = [];
4      state = automaton.initialState;
5      for event ← trace do
6          if state.canAccept(event) then
7              nextState = state.accept(event);
8              state = nextState;
9          else
10             if !state.HasIncompatibleTransition(event) then
11                 newState = state.addTransitionToNewState(event);
12                 state = newState;
13             else
14                 stateToModify, suffix = FindAncestorWithoutIncompatibleTransition(state,
                    event);
15                 for suffixEvent ← suffix do
16                     newState = stateToModify.addTransitionToNewState(suffixEvent);
17                     stateToModify = newState;
18                 end
19             end
20         end
21     end
22     return automaton
23 end
```

Algorithm 3: Pseudocode for CreateCompatibleAcceptor. This is a simplified version of the algorithm. In reality, the automaton is non-deterministic and given an event, there may be multiple transitions labeled with the same event.

10). If adding the transition results in an incompatible state (lines 13 - 19), we traverse backwards, looking for an ancestor where we can add transitions corresponding to the events up to e without introducing an incompatible state (line 13). From this ancestor node, we add new transitions and states to represent the events up to e (lines 15-18).

To find an ancestor from which it is possible to add a new chain of events such that an event can be added without incompatibility, we use the algorithm *FindAncestorWithoutIncompatibleTransition*. The details of *FindAncestorWithoutIncompatibleTransition* are given in Algorithm 4. We initialize the algorithm (lines 1-3) before we iteratively traverse the ancestors of the parent state. At this state of the DICE-Miner algorithm, all states have at most one incoming transition originating from another state, which is obtained using `getIncomingTransition`. As we traverse backwards, we collect the labels on the transitions (line 6). These labels correspond to the events that we will have to add transitions for. The traversal ends once we find an ancestor that we add the sequence of events including e without creating an incompatible state.

Algorithm 5 shows the check for an incompatible state. As we only add new outgoing transitions, it is sufficient to check pairs of the existing incoming transitions with the new event to detect NIF

Input: The current state, *state*, and the event that introduced an incompatible state, *event*
Output: An ancestor state and a sequence of events to add transitions for

```
1  parent = state;
2  suffix = [];
3  label = event;
4  while parent.HasIncompatibleTransition(label) do
5  |   grandParent, label = parent.getIncomingTransition();
6  |   suffix = label :: suffix;
7  |   parent = grandParent;
8  end
9  return parent, suffix
```

Algorithm 4: Pseudocode of FindAncestorWithoutIncompatibleTransition. Traverses the ancestry of a state to locate a state to branch from.

Input: A state, *state*, and an event to add, *event*
Output: true if adding the event does result in the state becoming incompatible

```
1  for transition ← state.incomingTransitions do
2  |   if NIF(transition,event) then
3  |   |   return true
4  |   end
5  end
6  if NF(state.prefix,event) then
7  |   return true
8  else
9  |   return false
10 end
```

Algorithm 5: Pseudocode of HasIncompatibleTransition. Checks if a transition labeled with the event can be added to the input state.

violations (lines 1-5). To check NF violations, we check the prefix of the state against the event (line 6). The prefix of each state is constructed by traversing all transitions on the trail of ancestor states. This includes all self-loops on each ancestor (i.e. the self-loops are traversed first before moving to a child state). It is enough to check for violations of NF and NIF, as the other properties are never violated in the first stage of DICE-Miner.

At the end of the first step, we receive an automaton that contains self-loops in some states, and these are the only cycles in it. The inferred model is a Non-Deterministic Automaton (NFA) as states can have multiple transitions with the same label. We presented simplified versions of the algorithms for ease for explanation. As the automata involved are NFAs, there may be more than one transition from a state given an event. The model is constructed based on the observed traces. It is sound with respect to these traces, and will accept all of these concrete traces.

3.5.2 Second Step. In the second step, we merge equivalent states in the automaton. To merge two states, a and b, we remove both states from the state machine, then add a new state, c. All the transitions from a and b are added onto state c. If there are multiple transitions with the same label, source, and destination, only one of them is kept and the duplicates are removed. As we already noted earlier, our model is non-deterministic and there may be multiple transitions from a state labelled with the same event.

To determine if two states are equivalent, we draw inspiration from the CONTRACTOR model [17] and define the equivalence of states based on the set of methods that are enabled. Concrete states that have the same enabled methods are merged. As described by de Caso et al. [17], this results in models with states that are intuitively interpreted and are at an abstraction level that developers find convenient. While we reuse the concept of the enabledness of methods in order to group states, our method is still primarily based on the execution traces that are input to DICE. The CONTRACTOR method requires further annotation and the computing of dependencies between the enabledness of all pairs of methods. In our work, we do not use these dependencies and other related concepts described by de Caso et al. [17] to avoid the need to annotate the pre- and post-conditions of every method. Prior work [33] has also shown that the performance of the CONTRACTOR approach is highly dependent on the quality of the pre- and post-conditions, and that it exhausts the running memory when provided with noisy invariants. Instead, we use the temporal properties to aid in determining the enabledness of a method at a particular state. We leave the study of the applicability of the dependencies of method enabledness on DICE and their integration into DICE for future work.

However, unlike de Caso et al. [17] and Krka et al. [33] which creates models from the state invariants of an object, we derive the enabledness of a method from the set of NF and NIF properties. If there are no LTL property that *disables* a method based on its prefix, the known incoming transitions, and the known outgoing transitions, then we consider that this method is enabled. We consider a method, say method A, to be *disabled* on a state from following conditions.

$$disabled(state, A) = \begin{cases} true & \text{if } NF(X, A), X \in prefix(state) \\ true & \text{if } NIF(X, A), X \in incoming(state) \\ true & \text{if } NIF(A, Y), A \text{ is pure}, Y \in outgoing(state) \\ false & \text{otherwise} \end{cases}.$$

If a method on a state leads to the automaton accepting a trace containing a pair of successive events violating the property, then the method is disabled. For example, a state with an incoming transition `add:true` (indicating that an item was successfully added) will have the event `isEmpty:false` disabled (the state can't be empty after a successful addition) by the second condition. As another example, a state with the outgoing transition `remove:true` (indicating that an item can be successfully removed from this state) will have the event `isEmpty:true` disabled (if an item can be successfully removed, it means it can't be empty) by the third condition. Methods can be enabled even if we have not seen the method invoked from a particular state in a concrete trace. Before merging two states, DICE-Miner checks that it does not lead to a violation of known LTL properties. Similar to the work by Lo et al. [41], a pair of states cannot be merged if it results in a state machine that violates any temporal property known to hold on the observed traces.

4 EVALUATION

To empirically evaluate our tool, we investigate 3 research questions.

- **RQ1: How effective is DICE in inferring FSA models?**
  RQ1 investigates the effectiveness of the adversarial specification mining approach by comparing the FSA models inferred by DICE against models inferred by state-of-the-art specification miners.
- **RQ2: How effective is DICE-Tester?**
  RQ2 investigates the effectiveness of the test generation component, DICE-Tester. This is done by comparing the tool against Evosuite, with DynaMOSA as its search algorithm, as a test generation baseline to answer this question. Instead of using the traces produced

by DICE-Tester, we study if using traces produced by Evosuite is enough to mine better specifications. Here, our objective is to evaluate the value of the temporal-property guided adversarial test generation strategy we built on top of Evosuite.

- **RQ3: How effective is DICE-Miner?**
  RQ3 investigates if the specification miner, DICE-Miner, can utilize the traces generated by DICE-Tester effectively. We compare DICE-Miner against DSM as a baseline, passing the same set of traces produced by DICE-Tester as input to both tools.

4.1 Experimental setup

4.1.1 Evaluation. To investigate these questions, we empirically evaluate the tools by assessing the quality of the models inferred against ground-truth models. Similar to previous studies, we measure precision and recall. This procedure for computing precision and recall has been used in prior studies [33, 35]. As input, a ground-truth model and the model inferred by DICE is provided. From these two models, traces are generated by randomly traversing the edges in the model. The precision of the inferred model is the percentage of traces produced by the inferred model that are accepted by the ground-truth model. The recall of the inferred model is the percentage of traces it accepts among the traces produced by the ground-truth model. In other words, precision is the proportion of traces from the inferred that are correct, and recall is the proportion of correct traces that the inferred model accepts. Finally, the quality of the model is measured using F-measure, computed as follows.

$$\text{F-Measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The 11 ground-truth models publicly released from Le et al.'s evaluation of DSM [36] are used [3]. These evaluation library classes, used for evaluating specification miners in previous studies, represent 100 analysed methods in total, and represent different categories of libraries ranging from data streaming to message exchange. To analyse the classes from JDK, we copied the source files of the corresponding classes from OpenJDK 1.8, to get around a constraint in Evosuite that prevented instrumentation and bytecode-rewriting of classes from some packages provided by the JDK. OpenJDK 1.8 was used as this was the version used in previous studies, and we observe that the choice of a more recent version will not impact the evaluation results as the ground-truth models involved only methods from earlier JDK versions. We also omitted traces from DICE-Tester containing events that are not present in the ground-truth models. This is done to allow direct comparison against the approaches used in previous studies. During our evaluation, we discovered a minor inaccuracy in the ground-truth model of ZipOutputStream, in which DICE-Tester found counterexamples to. Hence, we corrected the ground-truth model to account for the missing transition. Finally, for each case, we account for randomness by computing the average of the evaluation metrics from 20 runs of the experiment.

4.2 RQ1

To determine the effectiveness of our tool for answering RQ1, we compute precision, recall and F-measure of the output FSAs for 11 target library classes. We compare against DSM [36], which uses deep-learning and randomized test generation [47], as a baseline. For a second baseline, we also compare our tool against Tautoko [15], which leverages test generation to complete an initial FSA model. Tautoko takes the specifications inferred by the specification miner, ADABU [16]. Given an initial test suite, it learns an initial model using ADABU and then mutates test cases and executes them again to cover missing transitions in the initial model.

[3] https://github.com/lebuitienduy/DSM

Table 1. Precision, Recall, and F measure of DICE and DSM. NFST refers to NumberFormatStringTokenizer.

Class	DICE			DSM		
	P	R	F	P	R	F
ArrayList	31.4	**27.3**	**29.2**	**44.5**	16.3	23.9
HashMap	100.0	**94.1**	**97.0**	100.0	55.2	71.1
HashSet	**87.4**	**100.0**	**93.3**	74.0	62.4	67.7
Hashtable	84.0	**100.0**	**92.5**	**100.0**	66.6	79.9
LinkedList	100.0	**100.0**	**100.0**	100.0	23.7	38.4
NFST	**87.2**	**89.2**	**88.2**	54.1	70.2	61.1
Signature	100.0	**100.0**	**100.0**	100.0	91.2	95.4
Socket	**87.3**	**67.3**	**76.0**	58.4	62.6	60.4
StackAr	**86.8**	93.9	**89.8**	61.6	**97.1**	75.4
StringTokenizer	**100.0**	100.0	**100.0**	93.6	100.0	96.7
ZipOutputStream	**100.0**	100.0	**100.0**	80.6	100.0	89.3
Average	**87.8**	**88.3**	**87.8**	77.3	67.3	68.4

Table 2. Precision, Recall, and F-measure of Tautoko and DICE. NFST refers to NumberFormatStringTokenizer.

Class	DICE			Tautoko		
	P	R	F	P	R	F
ArrayList	31.4	27.3	29.2	-	-	-
HashMap	**100.0**	**94.1**	**97.0**	56.7	25.4	35.1
HashSet	**87.4**	**100.0**	**93.3**	100.0	13.6	23.9
Hashtable	**84.0**	**100.0**	**92.5**	38.8	23.3	29.1
LinkedList	100.0	**100.0**	**100.0**	100.0	20.9	34.6
NFST	87.2	89.2	88.2	**100.0**	**100.0**	**100.0**
Signature	100.0	**100.0**	**100.0**	100.0	23.8	38.4
Socket	**87.3**	**67.3**	**76.0**	84.1	24.4	37.7
StackAr	86.8	**93.9**	89.8	**100.0**	87.0	**92.8**
StringTokenizer	100.0	100.0	100.0	100.0	100.0	100.0
ZipOutputStream	100.0	**100.0**	**100.0**	100.0	25.0	40.5
Average	87.8	88.3	87.8	88.0	44.3	53.2

We initialized DICE using the test suite used by DSM in its evaluation, which is generated by Randoop [48]. From the results reported in Table 1, DICE improves on the average F-measure of DSM by over 19% (from 68.4 to 87.8), and, for every class, the difference between DICE and DSM is statistically significant – measured using the Wilcoxon signed-rank test. This indicates that DICE was effective in inferring FSA models.

We also investigated the effectiveness of DICE against Tautoko [15], which generates tests based on missing transitions in an initial model. We compare the FSAs mined by DICE against Tautoko's. The publicly available version of the executable artifact was downloaded from Tautoko's website[4] and executed on the same evaluation classes above, using Randoop generated tests as input to Tautoko. For some classes, Tautoko produced models containing methods that were not present in the ground-truth models. We therefore manually modified the models produced by Tautoko such that these methods were omitted, and merged states connected by a transition labelled with

[4]https://www.st.cs.uni-saarland.de/models/tautoko/

removed methods. For Socket, ZipOutputStream, and Signature, we evaluate the models published on Tautoko's homepage due to technical difficulties we encountered trying to run Tautoko on these classes. However, we modify the evaluation criteria as Tautoko does not produce models with transitions labelled with boolean return values of method calls. To compute the evaluation metrics for Tautoko's models, we ignore return values. This should generally lead to higher F-measures. We report the results in Table 2.

In one case (ArrayList), Tautoko does not run to completion within 24 hours. For the other classes that Tautoko can mine models for, we observe that Tautoko does not produce models of high F-measures. Apart from StackAr, DICE outperforms Tautoko on the 11 classes. While DICE produces models with an average F-measure of 87.8, Tautoko produces models with an average F-measure of 53.2. If we omit the model of ArrayList, then DICE produces models with an average F-measure of 93.7. We hypothesize that in certain cases, Tautoko's reliance on inspector methods (see Section 2.1.2) meant that it can not identify the right abstract states. For example, ZipOutputStream's state is not characterized by any inspector methods, and as such, Tautoko is unable to mine a good model of it.

The adversarial specification mining process implemented by DICE produces FSA-based models of higher quality, which outperforms existing approaches for inferring FSAs

4.3 RQ2

To answer RQ2, we aim to determine if the improvements was a result of our improvements to Evosuite, by studying if Evosuite alone was enough to produce diverse tests that would benefit the specification miner. We use Evosuite (version 1.0.6), with the DynaMOSA [50] search algorithm, as a baseline approach, collecting the traces produced by the final test suite that is the output of Evosuite. We use the default configuration of Evosuite. We do not try to find the optimal configuration for Evosuite as previous studies have indicated that tuning these parameters often do not outperform the default configuration [5]. By default, the population size of test cases is 50 individuals. The default crossover operator is used, which is the single-point crossover with probability of 0.75. The selection of test cases is done using tournament selection, with a tournament size of 10. Tests are mutated with a probability inversely proportional to the number of statements it contains. We use the same test budget of 15 minutes for both DICE-Tester and Evosuite. The traces produced from Evosuite are passed as input to DICE-Miner instead of the traces from DICE-Tester, and we compute the evaluation metrics of the FSA models produced.

The results are shown in Table 3. On some classes, DICE-Tester can outperform Evosuite by up to 18.9% in F-measure, and on average, DICE-Tester outperforms Evosuite by about 3.7%. To mitigate the effect of randomness, we run the experiments 20 times and compute if the differences are statistically significant using the Wilcoxon signed-rank test. We find that the differences are statistically significant for 4 out of the 11 classes in the benchmark. This indicates that, on its own, Evosuite is already effective in generating diverse test cases, although DICE-Tester can explore some uncommon usage patterns more effectively. We hypothesize that many of these classes do not exhibit complex usage constraints, therefore, Evosuite already performs well for these classes. However, to effectively explore non-trivial usage constraints, DICE-Tester can help significantly.

While Evosuite, with the DynaMOSA algorithm, is already able to aid the specification mining process by producing traces that allow DICE-Miner to achieve good performance, DICE-Tester takes it a step further, producing traces that are even better.

Table 3. Precision, Recall, and F-measure of DICE-Miner when using DICE-Tester and Evosuite. NFST refers to NumberFormatStringTokenizer. * indicate that the difference in F-measures is statistically significant.

Class	DICE-Tester			Evosuite		
	P	R	F	P	R	F
ArrayList *	31.4	**27.3**	**29.2**	**74.7**	17.0	27.6
HashMap	100.0	94.1	97.0	100.0	94.1	97.0
HashSet *	**87.4**	**100.0**	**93.3**	84.5	65.2	74.3
Hashtable	84.0	**100.0**	**92.5**	**91.0**	93.1	92.0
LinkedList *	100.0	**100.0**	**100.0**	100.0	89.9	94.7
NFST	87.2	89.2	88.2	87.2	89.2	88.2
Signature	100.0	100.0	100.0	100.0	100.0	100.0
Socket	**87.3**	**67.3**	**76.0**	86.4	67.5	75.8
StackAr *	**86.8**	**93.9**	**89.8**	68.9	84.7	76.0
StringTokenizer	100.0	100.0	100.0	100.0	100.0	100.0
ZipOutputStream	100.0	100.0	100.0	100.0	100.0	100.0
Average	87.8	**88.3**	**87.8**	**90.4**	81.9	84.1

Table 4. Precision, Recall, and F-measure of DICE and DSM with the traces produced from DICE-Tester. NFST refers to NumberFormatStringTokenizer.

Class	DICE			DSM		
	P	R	F	P	R	F
ArrayList	31.4	**27.3**	**29.2**	**60.4**	16.5	25.9
HashMap	**100.0**	**94.1**	**97.0**	30.8	86.0	45.3
HashSet	**87.4**	**100.0**	**93.3**	50.9	52.7	51.8
Hashtable	84.0	**100.0**	**92.5**	**93.3**	70.2	80.1
LinkedList	100.0	**100.0**	**100.0**	100.0	16.5	25.9
NFST	**87.2**	**89.2**	**88.2**	57.3	81.9	67.4
Signature	100.0	100.0	100.0	100.0	100.0	100.0
Socket	**87.3**	**67.3**	**76.0**	40.7	63.9	49.8
StackAr	**86.8**	93.9	**89.8**	47.2	**100.0**	64.1
StringTokenizer	**100.0**	100.0	**100.0**	75.3	100.0	85.9
ZipOutputStream	**100.0**	**100.0**	**100.0**	79.8	75.4	77.5
Average	**87.8**	**88.3**	**87.8**	64.7	71.4	63.4

4.4 RQ3

To answer RQ3, we compare DICE and DSM with both approaches using the same set of traces. We run DSM when provided with the traces from DICE-Tester. We compare the performance of DICE-Miner against DSM. In this study, we do not compare our tool against other specification miners since DSM [36] is the state-of-the-art specification miner and has been demonstrated to outperform multiple approaches such as the traditional k-tails algorithm, SEKT, TEMI and CONTRACTOR++ [36]. The results of using DSM with the additional traces are shown in Table 4.

We observe that DSM's performance does not improve with the additional traces. In fact, the additional traces causes the performance of DSM to decrease. As DSM's states are partially determined by the probability of observing each next transition, we hypothesize that it is sensitive to the set of input traces used and it does not handle low-probability, but still valid, transitions

robustly. Also noteworthy is that DICE-Miner can achieve a 100% F-measure in 4 of the 11 classes, while DSM achieves a perfect score in only one case. This shows that DICE-Miner is able to utilize diverse traces more effectively than DSM.

DICE-Miner outperforms a state-of-the-art specification mining technique even when the diverse traces produced by DICE-Tester are included in its input.

5 DISCUSSION

To further investigate our results, we raised four additional research questions for investigation and qualitatively analysed the models produced by DICE.

- **RQ4: How effective were our adaptations of Evosuite for finding counterexamples?**
  RQ4 studies the effectiveness of DICE-Tester in discovering counterexamples of temporal properties. Instead of measuring the quality of the FSAs output from DICE-Miner, an indirect measurement of how much the test generation benefited the specification mining process, we directly inspect the number of false temporal properties that the two tools, DICE-Tester and Evosuite (with DynaMOSA), are able to invalidate by finding counterexamples.
- **RQ5: How much did the constraints we introduced help to improve the performance of our specification mining algorithm?**
  RQ5 investigates the two constraints that we added in DICE-Miner motivated by the two observations we made. These observations are described in Section 3.5, involving method purity and incompatible transitions on a state. The first constraint ensures that transitions labelled with side-effect free methods are self-loops, and the second constraint prevents the erroneous conflation of states that may accept traces that violate previously-mined temporal properties. We try to drop these constraints and observe their effect on the performance of DICE-Miner.
- **RQ6: How much does the quality of the initial test suite affect DICE?**
  RQ6 varies the quality of the initial test suite by using reduced subsets of it as input to the DICE process. We aim to investigate if the quality of the initial test suite has an effect on the quality of the models inferred by DICE.
- **RQ7: Can the FSAs inferred by DICE be used to support additional testing activities, for example, to perform protocol fuzzing?**
  RQ7 investigates if the FSAs learned by DICE can be used to aid in fuzzing servers of stateful protocols. Effective fuzzing of a server requires the fuzzer to be aware of the specific order of messages to reach certain states. We use state models learned by DICE to initialize a server fuzzer and observe if it helps the fuzzing process. This functions as an evaluation of DICE to determine if its inferred models have practical applicability.

5.1 RQ4

To try to further quantify the difference in performance of DICE-Tester and Evosuite+DynaMOSA, we compare the set of temporal properties that were successfully invalidated at the end of the test generation process. We evaluate them against the ground-truth properties annotated by human experts. These ground truth properties were made publicly available by Sun et al. [61]. The human experts annotated each possible temporal property following the template indicating if the property is true. This was done for three classes, HashSet, StringTokenizer and StackAr, which we use in our evaluation.

As input to DICE-Tester, we enumerate all possible temporal properties between the methods of the class, and run DICE-Tester. For Evosuite+DynaMOSA, during the test generation process, we collect the traces when executing tests and print to standard output if the trace of the test produced

Table 5. Number of incorrect rules failed to be invalidated when using all possible LTL properties as input

Class	DICE-Tester	Evosuite
HashSet	**51**	233
StackAr	**39**	108
StringTokenizer	**35**	216
Average	**41.7**	185.7

Table 6. Precision, Recall, and F-measure for DICE varying the threshold for resetting the test population. NFST refers to NumberFormatStringTokenizer.

Class	**100**	**50**			**150**		
	F	**P**	**R**	**F**	**P**	**R**	**F**
ArrayList	**29.2**	72.6	15.7	25.8	78.0	16.1	26.6
HashMap	97.0	100.0	94.1	97.0	100.0	100.0	**100.0**
HashSet	**93.2**	77.9	18.3	30.0	95.9	80.7	87.6
Hashtable	92.5	98.2	100.0	**99.1**	96.6	100.0	98.3
LinkedList	100.0	5.9	44.4	10.4	100.0	100.0	100.0
NFST	**88.2**	87.7	86.4	87.0	86.6	85.6	86.1
Signature	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Socket	**76.0**	53.4	66.0	59.0	80.5	66.0	72.5
StackAr	**89.8**	41.7	88.2	56.6	83.2	84.7	84.0
StringTokenizer	100.0	100.0	100.0	100.0	100.0	100.0	100.0
ZipOutputStream	**100.0**	28.6	100.0	44.4	72.7	100.0	84.2
Average	**87.8**	69.6	73.9	64.5	91.3	86.7	86.7

is a counterexample to a temporal property. Note that the set of input temporal properties will include properties that contradict each other (e.g. both `NF(A, B)` and `AF(A, B)` are part of the input). In total, there were 56 true properties for HashSet out of 1014 possible properties, 35 true properties out of 384 possible properties for StringTokenizer and 42 true properties out of 600 possible properties for StackAr.

The results are reported in Table 5, and we observe that in each class, DICE-Tester successfully found counterexamples for a vast majority of the incorrect properties. Out of an average of 621.7 incorrect properties, DICE-Tester successfully constructs tests that contradict an average of 580 of them, or over 93% of the them. In contrast, Evosuite does not succeed in invalidating most of the incorrect properties, and there were four times the number of incorrect properties that Evosuite failed to find counterexamples of.

Next, we also investigate the effect of the threshold for resetting the test population, described in Section 3.3. On average, the test budget of 15 minutes allows to search to run for 505 generations. With a threshold of 100 generations, we observe an average of 1 reset for each run. We varied the threshold and run DICE on the classes in the benchmark. The evaluation metrics are reported in Table 6. Our findings are that having too low a threshold adversely affects the quality of the models learned by DICE. Decreasing the threshold from 100 generations to 50 generation caused a large drop in F-measure from 87.8% to 64.4%, over a 20% decline. On the other hand, increasing the threshold to 150 generations caused a slight decrease in quality, from 87.8% to 86.7%. The results suggest that, at least for this benchmark set of classes, the default value that we choose (100 generations) is reasonably good.

Table 7. F-measure of DICE-Miner, without the constraints we identified. NFST refers to NumberFormat-StringTokenizer.

Class	Both	Constraint A, w/o B	Constraint B, w/o A	None
ArrayList	29.2	28.6	30.4	31.3
HashMap	97.0	97.0	97.8	97.8
HashSet	93.2	88.5	91.8	92.3
Hashtable	92.5	86.3	84.9	87.7
LinkedList	100.0	100.0	70.8	70.8
NFST	88.2	72.7	81.8	81.8
Signature	100.0	100.0	100.0	100.0
Socket	76.0	63.5	67.9	67.8
StackAr	89.8	71.9	65.2	74.5
StringTokenizer	100.0	100.0	100.0	100.0
ZipOutputStream	100.0	100.0	100.0	100.0
Average	87.8	82.6	80.9	82.2

DICE-Tester outperforms Evosuite in finding counterexamples for incorrect temporal properties, validating the benefits of our modifications for improving the search process.

5.2 RQ5

To study if the two constraints described in Section 3.5 influenced the performance of DICE-Miner, we ran more experiments, omitting the constraints. Constraint A refers to the constraint that pure methods do not cause a transition to another state, while Constraint B refers to the constraint that a state should be split up if it has incompatible transitions. We use the execution traces from Le et al. [36] and DICE-Tester in these experiments.

The results are presented in Table 7. Without both constraints, the average F-measure dropped by about 5.2%. In virtually all classes, we see a decline in the performance of DSM-Miner. We see that using information about method purity is important to achieving good performance. While using Constraint B alone did not provide any improvements without Constraint A, it was necessary to increase the performance to 87.8 from 82.6 (with only Constraint A). This confirms our observations that these two constraints are important for specification mining.

The two constraints that we added for addressing the two observations were helpful for DICE-Miner to achieve its performance.

5.3 RQ6

To answer RQ6, we investigate how sensitive the DICE process is to the initial input test suite. We performed experiments using different subsets of the initial test suite. We do not run experiments using the test suite originally written by the developers accompanying the classes in the benchmark. We manually analysed the test cases for these classes and found that exercising these tests would only produce a few traces. The functionality of each class is well tested, but the tests typically do not have a high diversity regarding the sequences of method invocations. For example, the test cases for a LinkedList exercise each method of the LinkedList, but do not show how the methods relate to one another. Therefore, we do not expect the evaluation metrics of DICE to change significantly from using a small subset (e.g. 25%) of the initial test suite used as input to the earlier experiments.

Table 8. Precision, Recall, and F-measure for DICE varying the initial test suite. NFST refers to NumberFormatStringTokenizer.

Class	100%	50%			25%		
	F	P	R	F	P	R	F
ArrayList	**29.2**	31.9	21.1	25.4	28.6	25.3	26.8
HashMap	97.0	100.0	94.1	97.0	100.0	83.1	90.8
HashSet	93.2	92.4	100.0	96.0	94.1	100.0	**97.0**
Hashtable	92.5	98.0	100.0	99.0	96.3	100.0	**98.1**
LinkedList	100.0	100.0	100.0	100.0	100.0	100.0	100.0
NFST	88.2	92.1	89.4	90.7	92.9	87.9	**90.3**
Signature	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Socket	**76.0**	84.8	62.5	72.0	88.3	61.7	72.6
StackAr	**89.8**	76.4	81.9	79.0	73.9	84.7	78.9
StringTokenizer	100.0	100.0	100.0	100.0	100.0	100.0	100.0
ZipOutputStream	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Average	**87.8**	88.7	86.3	87.2	88.6	85.7	86.8

The results are shown in Table 8. By using 50% of the initial test suite, F-measure drops from 87.8% to 87.2%, and by using 25% of the initial test suite, it drops further to 86.8%. This indicates that the initial quality of the test suite influences the quality of the models inferred by DICE, however, the difference is small. Interestingly, we noticed that F-measure increased for some of the classes in the benchmark. The models for ArrayList and HashSet improved in quality by about 1% in F-measure. For these cases, as having fewer initial traces may mean that we infer a larger number of incorrect temporal properties, we hypothesize that these large number of incorrect traces can sometimes lead the search process to collaterally cover more temporal properties and produce informative traces that were useful to the inference process. We answer RQ6 by concluding that the quality of the initial test suite has an impact on the models inferred by DICE, but overall, this impact is small.

The quality of initial test suite has a small effect (1% change in the average F-measure) on the quality of the models inferred by DICE.

5.4 Qualitative Evaluation

While the DICE system results in an improved F-measure compared to existing approaches, it is not able to achieve 100% correct finite-state automata for all of the 11 classes. We manually inspected the resulting FSA to try to identify reasons as to why this is the case, and to propose next steps for our work.

One interesting observation is that the interpretation of FSA models may differ between models that have identical F-measures. Apart from boosting the accuracy of models, having knowledge of pure methods will produce models that are qualitatively better for program comprehension. For example, referring to Figure 10, DSM produces a model for StringTokenizer that suggests that `nextToken` is always enabled should a developer never invoke `hasMoreTokens`. In contrast, the model produced by DICE-Miner, as shown in Figure 11, cannot be erroneously interpreted in this manner. The invocation of `hasMoreTokens` with different return values are not allowed on the same state and it is clear that `nextToken` may change the state of a StringTokenizer object such that `nextToken` can not be further invoked. Note that both the models produced by DSM and

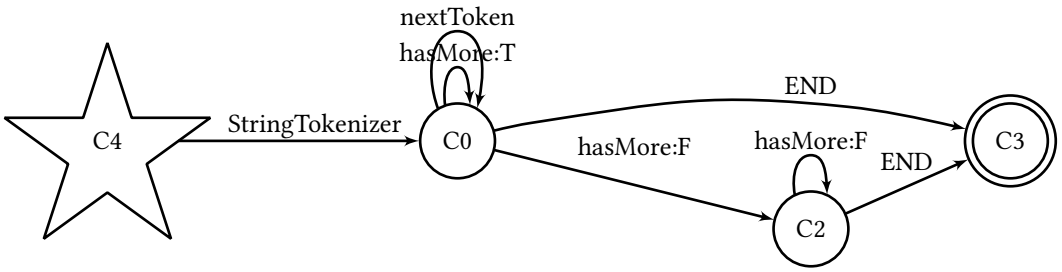

Fig. 10. Example of a FSA model produced by DSM. hasMore:T is short for hasMoreTokens:TRUE and hasMore:F is short of hasMoreTokens:FALSE

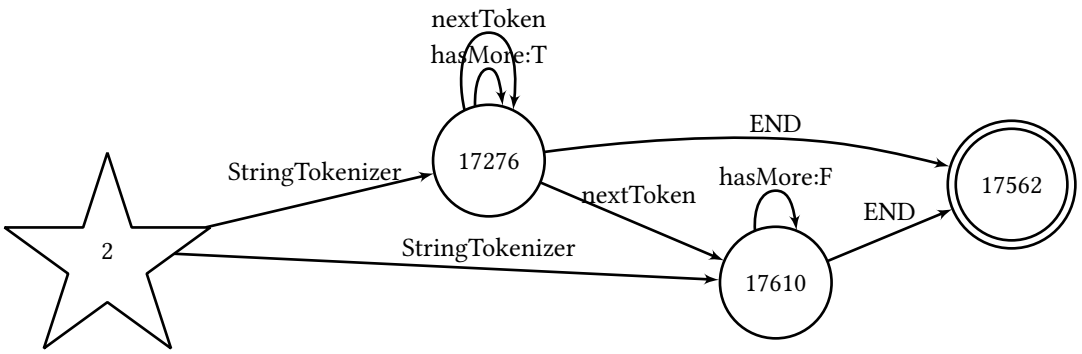

Fig. 11. Example of a FSA model produced by DICE-Miner. hasMore:T is short for hasMoreTokens:TRUE and hasMore:F is short for hasMoreTokens:FALSE

DICE-Miner have an F-measure of 100, showing that there may be qualitative differences between models that are automatically evaluated to be perfectly accurate.

The method `hasMoreTokens` represents an interesting case that may be a next step for DICE-Miner. While it is accurately identified as a side-effect free method heuristically by DICE-Miner, the static analysis we performed did not reveal it as a pure method. The implementation of `hasMoreTokens` is, in fact, impure. A static analysis-based approach therefore treats hasMoreTokens as an impure method. More sophisticated analysis of purity, such as using the notion of observational-purity [6, 46] will help to produce qualitatively better models. Observationally-pure methods refer to the class of methods, in which `hasMoreTokens` belongs to, which have side-effects that cannot be observed outside of the class. From the perspective of learning usage models and specifications about usage of the class, these methods are effectively pure. While our naive name-based heuristic may help to identify some of these methods, it is likely that there are observationally-pure methods in the wild that we cannot detect.

Investigating the poor performance of DICE for some classes, the lack of expressiveness of the six LTL property templates considered is a possible reason for it. LTL properties templates involving more than 2 events and the use of other temporal operators beyond the basic operators may be required. For example, clients of NumberFormatStringTokenizer are able to reset its state using the `reset` method, as such, there are constraints between its methods that can only be expressed through the use of LTL formula involving more than 2 events. The property

`NF(hasNextToken():FALSE,nextToken)` is false, contrary to intuition, since invoking `reset` after `hasNextToken:FALSE` may allow `nextToken` to be invoked again. In this case, the three-event property `(hasNextToken():FALSE NF nextToken) U reset` is necessary to accurately represent the temporal constraints between the methods. This formula indicates that `nextToken` cannot follow `hasNextToken():FALSE` until the object instance has been `reset`.

While in this work, we have considered only 2 event LTL formulae, it may be possible to use formulae relating 3 or more events and use them during the testing process or to guide the merging of the states in the automaton. However, having more complex formulae will come at a cost and there may be a trade-off; gaining some accuracy but slowing down the approach. Including longer rules will lead to an exponential growth in their numbers. Indeed, many studies have focused on mining rules and patterns involving only 2 events [35, 40, 59, 66], and researchers have noted the problem of scalability when mining longer patterns [52].

5.5 RQ7

5.5.1 Using the inferred FSAs for fuzzing. Next, we evaluate models produced by DICE in its applicability on fuzzing servers of stateful network protocols [54]. In fuzzing, random test inputs are generated automatically in order to find bugs. It is important to discover critical bugs in the implementation of protocols. The Heartbleed vulnerability [5], a security bug in the implementation of the Transport Layer Security in the OpenSSL library, has shown the pervasiveness and the high cost of such bugs [19]. Server fuzzing may help in finding these bugs and several server fuzzers, such as AFLNET [54], have been proposed. Servers are stateful and reaching specific states may require specific sequences of messages between server and client. Without information about the specific messages required, the fuzzer is unlikely to send a sequence of messages that exercises program states deep in the server. The FSA model learned by DICE can be used as a state model to guide the server in reaching states that are difficult to reach otherwise.

AFLNET [54] is a coverage-guided fuzzer that has been shown to outperform other server fuzzers. Like other coverage-guided fuzzers, AFLNET instruments the program to receive feedback if there is an increase in the server's code coverage achieved by an input. This feedback is used by the fuzzer to decide which inputs to mutate, optimising its choice of inputs for increased coverage of the program. Unlike other coverage-guided fuzzers, AFLNET has a state model of the server and detects if inputs lead to unexplored states. In other words, an input is retained for further mutation by AFLNET if it leads to increased coverage or enters an unexplored state. During the fuzzing process, AFLNET constructs a state model of the server using the observed status code sent by the server. By updating the state model during runtime, AFLNET is able to detect bugs in states that only appear in the implementation of the protocol and not in the protocol's official definition. As a tradeoff, AFLNET may spend a significant amount of time early in the fuzzing process using a rudimentary state model and, until the state model is refined, is unable to reach deep into the program.

During the fuzzing process, AFLNET selects the next input for fuzzing by selecting a state in the state model to act as the target state. From this target state, two criteria are applied. First, AFLNET picks, with a higher probability, states that are less frequently exercised. Second, states that have been associated with a greater number of inputs that lead to higher coverage or new states are more likely to be selected. Then, an input associated with that state is mutated in a way that ensures that that particular state is still visited by the new input.

A state inference algorithm, such as DICE, can enhance the workings of AFLNET by providing it with the state model before starting the fuzzer. By running DICE on a protocol client, DICE produces

[5] https://heartbleed.com/

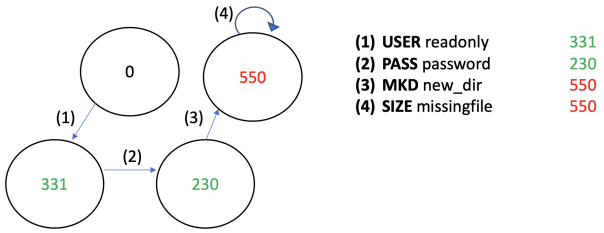

Fig. 12. State traversal in AFLNET's inferred state machine for the given sequence of request types

a model of the protocol which can be used for server fuzzing. We observe that the API members of a protocol's client tend to have a one-to-one correspondence to the request types defined in a protocol. For instance, an FTP client typically has API members for each request type (e.g. the `user()` method sends a USER request, the `pass()` method sends a PASS request). Consequently, **an automaton specification of a protocol's client API is a model of the protocol from the client's perspective**. The state model is, therefore, useful for a fuzzer to simulate the protocol's client. To target a particular state, one can traverse the edges from the starting state until the target state is reached; the labels on the edges traversed are the requests the client will have to make.

We modify the state-of-the-art server fuzzer, AFLNET, to use the automaton produced from DICE as the state model. Instead of starting the fuzzing process with an empty state model, we modify it to be initialized with the output of DICE. We also make some modifications in AFLNET related to how it uses a state model. Presently, the status code in the server's response is used as an indicator of the present state. On the other hand, the states in DICE's state machine are more granular. We modify AFLNET to traverse the state machine while considering more information than the response code of the server, namely, the sequence of requests made so far. Additionally, when the server responds with a status code indicating an error, we assume that the state has not changed. In contrast, AFLNET transits to an error state when the server replies with server code indicating an error. Overall, with our modifications and initialization with DICE's output, AFLNET can work with a higher granularity of states and is more likely to generate inputs that are accepted, allowing it to generate inputs that go deeper into the program. To distinguish between the two systems, we will refer to our modified version of AFLNET as DICE+AFLNET.

To look at an example of the difference between AFLNET and DICE+AFLNET, we use the following example sequence of requests made by AFLNET: `[USER, PASS, MKD, SIZE]`. In this sequence, the client has successfully logged in with a user. The user then fails to create a new directory, due to insufficient access rights, and attempts to get the size of file that does not exist on the server. Given this sequence of requests, the server responds with the following sequence of response codes: `(331, 230, 550, 550)`, indicating that the login was successful but the creation of the new directory and access of a non-existent file have failed. AFLNET traverses the states as shown in Figure 12, in which it first moves to state 331, then to 230, and to 550. On the other hand, DICE+AFLNET traverses the states as shown in Figure 13. Notice that DICE+AFLNET remains in the same state after the failed requests. Compared to the transition to state 550 used in AFLNET, we suggest that this more accurately captures the semantics of a failed request in the two protocols that we study. As this may not be true of many network protocols, we will provide configuration options allowing the user of the fuzzer to specify the semantics of a failed request with respect to the state model. Overall, this enables AFLNET to work with the state models from DICE, which are more granular.

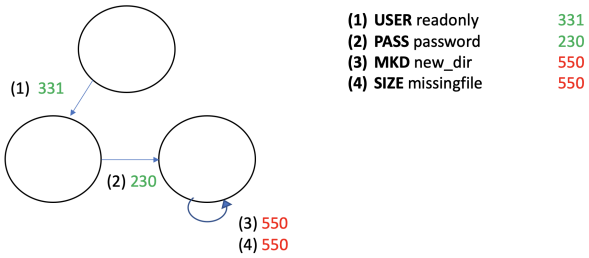

Fig. 13. State traversal in DICE's inferred state machine for the same sequence of request types as Figure 12

5.5.2 Experimental Results. To determine if the models learned by DICE can aid server fuzzers, we evaluate DICE+AFLNET on two protocols that were studied in the evaluation of AFLNET [54], FTP and RTSP. We reuse the same FTP[6] and RTSP[7] servers that were fuzzed by Pham et al. [54]. We run the modified version of AFLNet that takes a state machine from DICE as input. For each of the two protocols, we performed a search on GitHub to find open-source clients[8][9] of the protocol. We selected one client for each protocol and ran DICE on it. This process is semi-automatic. Some human effort was required to annotate which API methods directly represent a request from the client. We modified the source of the clients to enable DICE to run effectively on it. Originally, the clients have different API usage patterns in which the developers have to check the integer return value of the method call (`int returnCode = ftpClient.user(username)`), or invoke another method to retrieve the response server code (`if (parseServerResponse() != 200)`). We modified the clients to throw exceptions on requests where the server responds with status codes signalling failure. For each of the two clients, the modification of the client took less than 15 minutes. This was done for DICE to detect a failing method call. DICE was run on each client for another 15 minutes.

The modification of the client and annotation of the methods took less than 15 minutes for each protocol. In total, less than 30 minutes is required to make the necessary modifications and to run DICE. For a fair comparison, we grant AFLNET an additional hour of fuzzing to account for the additional effort and time to run DICE. We fuzz each protocol for 24 hours using DICE+AFLNET, and 25 hours using the original version of AFLNET. We compare the fuzzers by the average line and branch coverage of the resulting inputs. Apart from the additional step of running the automata inference algorithms and the different running time, we follow the same fuzzing procedure as prior work. As the fuzzing process is stochatic, we mitigate the effect or randomness by running 20 independent experiments for each fuzzer and report the average coverage obtained.

The average coverage achieved by the fuzzers are shown in Table 9. On the RTSP server, there was only a slight increase in coverage (of 17 lines and 18 branches). As Pham et al. [54] observed, the RTSP server has fewer states to explore and the number of messages required to exercise code deeper into the server is lower than FTP. In other words, for such protocols where the state model is simpler, AFLNET benefits less from having a state model initialized before it begins fuzzing. On the other hand, on the FTP server, the use of the state machine from DICE significantly improves

[6]https://github.com/hfiref0x/LightFTP
[7]http://www.live555.com/mediaServer/
[8]https://github.com/apache/commons-net
[9]https://github.com/mutaphore/RTSP-Client-Server

Table 9. Coverage achieved on each protocol server by the AFLNET and DICE+AFLNET. Numbers in parenthesis indicate the proportion of total lines/branches covered.

Protocol	AFLNET		DICE+AFLNET	
	Lines Covered	Branches Covered	Lines Covered	Branches Covered
FTP	644 (57%)	311 (40%)	**777 (69%)**	**400 (50%)**
RTSP	2453 (11%)	1216 (7%)	**2470 (11%)**	**1234 (7%)**

Table 10. Coverage achieved on each protocol server when using the state models from DSM and DICE. Numbers in parenthesis indicate the proportion of total lines/branches covered.

Protocol	DSM+AFLNET		DICE+AFLNET	
	Lines Covered	Branches Covered	Lines Covered	Branches Covered
FTP	703 (62%)	341 (43%)	**777 (69%)**	**400 (50%)**
RTSP	2448 (11%)	1201 (7%)	**2470 (11%)**	**1234 (7%)**

both line and branch coverage. The average line coverage increased from 57% to 69% and branch coverage increased from 40% to 50%.

Next, we investigate if AFLNET achieves the same increase in performance when we use the automata inferred by DSM [36] instead of DICE. The experimental results are shown in Table 10. On the RTSP server, the difference between the use of models from DSM and DICE is insignificant. However, on the FTP server, DICE achieves a significantly higher line and branch coverage. We also observe that the coverage on the FTP server increased over the baseline AFLNET fuzzer; it increases from 57% to 62% line coverage, suggesting that the modifications we made to initialize the fuzzing with an initial state machine were useful.

Based on the improvements on fuzzing the FTP server, we answer RQ7 by concluding that there is evidence that the models learned by DICE have practical applicability to a downstream application. DICE provides an informative state model that the fuzzer can use to guide the mutation of inputs, and may effectively boost the performance of a stateful server fuzzer. In the future, we will further evaluate DICE on more protocols.

The FSA models learned by DICE have practical downstream application and can be used for initializing a fuzzer with a state model. It improves the coverage achieved by a fuzzer on the servers of a two stateful protocols.

5.6 Threats to Validity

Threats to internal validity. In our studies, we have tried to reuse most of existing implementation whenever possible. While it may be possible that there are bugs in the code we have written, we have checked them multiple times to reduce threats to internal validity. We have evaluated DICE in different aspects, such as evaluating that each component in DICE outperforms strong baselines (DSM and Tautoko for DICE, Evosuite for DICE-Tester, DSM for DICE-Miner). We have performed a deep analysis, including a qualitative analysis, for a deeper understanding of DICE.

Threats to construct validity. In our experiments, we used common evaluation metrics that were used in previous studies. The evaluation process for computing these metrics has been used in previous studies and is well-understood.

Threats to external validity. While it may be possible that our findings do not generalize to other library classes and APIs, we have considered the 11 classes evaluated in previous studies on specification mining. These classes are diverse, coming from both the Java standard library

and other third-party libraries. Furthermore, these are classes used in real-world software and are from a range of different domains. We emphasize that our approach cannot capture every possible constraint of an API or an object class, and it may not be possible to propose search goals for every constraint. The models DICE learned may not always be a realistic representation of every program. DICE is currently limited to mining finite-state automata, corresponding to regular languages. Still, on a benchmark created by prior studies, we have shown that the automata mined by DICE are more accurate specifications than those mined by prior approaches. Although DICE cannot mine specifications represented by a context-free language, we note many specifications mined in the literature are regular languages and researchers have found uses for these specifications, such as analysing and finding security flaws in bank cards [1] and TLS [18], or modelling Android applications [56]. Moreover, as we have shown in Section 5.5, DICE can still learn models that are useful on a downstream task. We leave the mining of other types of specifications, such as those equivalent to context-free languages, as future work.

6 RELATED WORK

We have provided an overview and explained the background of our study in Section 2, therefore we limit the discussion in this section. A key difference between our work and previous studies on specification mining is that we rely on an adversarial test generation process to prune incorrect properties. For FSA inference, our work differs from existing work as we incorporate knowledge of method purity and do not make the assumption that states with the same prefix can be merged.

Usage models have been incorporated in test case generation previously by Fraser and Zeller [26]. However, they use usage models to improve the readability of test cases by guiding test generation to resemble test cases written by humans, and reducing the amount of nonsensical test cases generated. In contrast, our work aims to generate test cases that are correct, but do not resemble common usage patterns that tend to be exercised by existing tests.

For the generation of test inputs, various approaches first learn the probability distribution of observed test inputs. While most techniques generate test inputs that resemble the learned distribution to produce synthetically correct inputs, the Skyfire [64] approach learns a probability distribution to generate test inputs that do not resemble the examples it has seen, applying heuristics such as favouring low probability rules. Pavese et al. [51] proposed inverting the probabilities in grammar-based test generation to explore uncommon test inputs. Our work shares a similar goal with these two studies, aiming to explore uncommon behavior in testing. However, our study focuses on a different domain of temporal specifications instead of input generation. Instead of using a probability distribution for modelling test inputs, we use a set of LTL property templates to characterize the execution traces while running test cases.

There has been many studies on search-based testing. Studies have shown that a multi-target formulation of code coverage outperforms a single-target formulation for test generation [49]. As discussed earlier in Section 2, Panichella et al. [50] proposed the DynaMOSA algorithm that enables dynamic selection of targets in the multi-objective problem of test case generation. They modelled the dependencies between structural goals, allowing it to outperform alternative search algorithms for search-based testing.

Many studies have studied diversity in test generation [11, 23, 24, 43]. For example, researchers have studied the diversity of test inputs and outputs [2]. Shin et al. [60] have proposed the use of diversity for mutation testing. Our work differs from these studies as we focus on the diversity of execution traces produced by tests, instead of modelling the diversity of the inputs or outputs of each method. Moreover, we use test generation only to produce more diverse traces, supporting our objective of learning accurate FSA specifications.

There are other studies on the diversity of software traces, for example, coverage information collected from test execution [24, 39]. Typically, these studies propose metrics over traces to measure the similarity of tests to maximize fault detection capability [23]. Our goal in this study is different, aiming to diversify traces for mining more accurate FSA specification models of a software system.

DICE is a counterexample-driven approach that has similarities to approaches based on Angluin's L* algorithm [4, 31]. In these studies, the oracle (aka. Minimally Adequate Teacher) provides the learner with a counterexample to answer equivalence queries. Using the counterexample, the learner refines the model further. In DICE, the DICE-Tester component can be viewed as a component providing counterexamples of LTL formulae, which is used later to construct the automata. There are similarities in how counterexamples to a specification (LTL formulae in DICE, the automata in L*) is used to improve the model. However, unlike the Minimally Adequate Teacher, the DICE-Tester is itself unaware of the ground-truth model, relying on a search-based algorithm to falsify temporal properties. DICE is not an active algorithm as there is no oracle interacting with the system. Therefore, compared to approaches based on Angluin's L* algorithm, DICE is more practical and can work for software systems without an oracle.

There are also similarities between DICE and model checking approaches that leverage counterexamples. One example is CEGAR [13], that performs counterexample-guided abstraction refinement during model checking. In CEGAR, counterexamples are used to split the abstract state as a counterexample indicates that there is some behaviour in the abstract model that is not present in the concrete version. The abstract state is split such that it no longer admits the counterexample. DICE is different from these approaches as it does not perform model checking, but it infers a model from a concrete system.

Adaptive model checking [28] is akin to black-box model checking [53], where there is no initial model of the system. In adaptive model checking, an inaccurate model is updated as it is used to verify a software system. Inaccuracies in the model may be due to differences caused by updates to the system. Like other Angluin-style automata learners, counterexamples are used to incrementally improve the model. These model checking techniques use the Vasilevskii-Chow algorithm [12, 63] for conformance testing to check if the model is equivalent to the system. DICE is similar as it tests the system against specifications, but differs in that it never checks or verifies conformance between the model and software system, which can be expensive and impractical with a cost exponential to the size of the automaton. DICE avoids this cost, only performing search-based testing to search for traces that falsify individual LTL properties.

7 CONCLUSION AND FUTURE WORK

To conclude, we proposed a new approach of adversarial specification mining and prototyped a tool, DICE (**Di**versity through **C**ounter-**E**xamples), for mining specifications. DICE systematically diversifies execution traces and addresses shortcomings in current specification mining algorithms. By adversarially guiding test generation towards finding counterexamples of the specification, our approach produces diverse traces that represent uncommon but correct usage of the program. To do so, we introduce new fitness goals representing counterexamples to temporal specifications expressed in LTL properties, address shortcomings in the LTL property templates used in previous studies, and use search-based testing to produce diverse traces. To take advantage of the diverse traces and the temporal properties, we propose a new specification mining algorithm that utilizes knowledge of method purity and use the temporal specifications to prevent erroneous merges to infer Finite-State Automata models with improved precision and recall. Finally, in our empirical evaluation, our approach significantly outperforms DSM, the current state-of-the-art specification miner, and Tautoko, which generates tests for specification mining. DICE produces models with an average F-measure of 87.8, while the current state-of-the-art approach, DSM, produces models

with an average F-measure of 68.4. and Tautoko produces models with an average F-measure of 53.2 in our experiments. Furthermore, our experiments suggest that the performance of DSM does not always improve when provided with more data. The artifact website of DICE can be found at https://kanghj.github.io/DICE

While we focus on generating uncommon sequences of method invocations in this study, we hope to explore the integration of methods that diversify test inputs [11, 23] to improve DICE's ability to generate uncommon test cases in future. We also hope to investigate more expressive LTL property types and evaluate DICE with other specifications beyond those that were studied in prior work. We will also study the tradeoffs of including longer temporal properties in future. Another possible direction is to explore more complex properties using temporal properties that were hard for DICE-Tester to falsify. The difficulty in falsifying them may indicate that the properties hold, or that there are more complex relationships between the events in the property. Users of DICE may also find these properties useful. We will also study other ways to improve the effectiveness of DICE-Tester. To that end, we hope to explore the use of techniques such as Swarm Testing [29], which may help to further increase the diversity of tests.

REFERENCES

[1] Fides Aarts, Joeri De Ruiter, and Erik Poll. 2013. Formal models of bank cards for free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 461–468.

[2] Nadia Alshahwan and Mark Harman. 2012. Augmenting test suites effectiveness by increasing output diversity. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1345–1348.

[3] Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16.

[4] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106.

[5] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18, 3 (2013), 594–623.

[6] Mike Barnett, David A Naumann, Wolfram Schulte, and Qi Sun. 2004. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on formal techniques for Java-like programs (FTfJP)*.

[7] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D Ernst, and Arvind Krishnamurthy. 2014. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering* 41, 4 (2014), 408–428.

[8] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 267–277.

[9] Nimrod Busany and Shahar Maoz. 2016. Behavioral log analysis with statistical guarantees. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 877–887.

[10] Zherui Cao, Yuan Tian, Tien-Duy B Le, and David Lo. 2018. Rule-based specification mining leveraging learning to rank. *Automated Software Engineering* 25, 3 (2018), 501–530.

[11] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. 2010. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software* 83, 1 (2010), 60–66.

[12] Tsun S. Chow. 1978. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering* 3 (1978), 178–187.

[13] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)* 50, 5 (2003), 752–794.

[14] Hila Cohen and Shahar Maoz. 2015. Have We Seen Enough Traces?(T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 93–103.

[15] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. 2010. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 85–96.

[16] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. 2006. Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*. ACM, 17–24.

[17] Guido de Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. 2010. Automated abstractions for contract validation. *IEEE Transactions on Software Engineering* 38, 1 (2010), 141–162.

[18] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of {TLS} Implementations. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 193–206.
[19] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*. 475–488.
[20] Matthew B Dwyer, George S Avrunin, and James C Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*. IEEE, 411–420.
[21] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 73–84.
[22] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.
[23] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 223–233.
[24] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. 2008. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. IEEE, 178–186.
[25] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 416–419.
[26] Gordon Fraser and Andreas Zeller. 2011. Exploiting common object usage in test case generation. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 80–89.
[27] Gregory Gay. 2017. The fitness function for the job: Search-based generation of test suites that detect real faults. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 345–355.
[28] Alex Groce, Doron Peled, and Mihalis Yannakakis. 2002. Adaptive model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 357–370.
[29] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 78–88.
[30] Wei Huang and Ana Milanova. 2012. ReImInfer: method purity inference for Java. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 38.
[31] Hardi Hungar, Oliver Niese, and Bernhard Steffen. 2003. Domain-specific optimization in automata learning. In *International Conference on Computer Aided Verification*. Springer, 315–327.
[32] Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press.
[33] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 178–189.
[34] Tien-Duy B Le, Xuan-Bach D Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing specification miners through model fissions and fusions (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 115–125.
[35] Tien-Duy B Le and David Lo. 2015. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 331–340.
[36] Tien-Duy B Le and David Lo. 2018. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 106–117.
[37] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 602–613.
[38] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL specification mining (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 81–92.
[39] David Leon and Andy Podgurski. 2003. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* IEEE, 442–453.
[40] David Lo, Siau-Cheng Khoo, and Chao Liu. 2008. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 20, 4 (2008), 227–247.

[41] David Lo, Leonardo Mariani, and Mauro Pezzè. 2009. Automatic steering of behavioral model inference. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 345–354.
[42] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*. ACM, 501–510.
[43] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. 2018. Fast approaches to scalable similarity-based test case prioritization. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 222–232.
[44] Urko Rueda Molina, Fitsum Kifetew, and Annibale Panichella. 2018. Java unit testing tool competition-sixth round. In *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 22–29.
[45] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: Why do Java developers struggle with cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 935–946.
[46] David A Naumann. 2007. Observational purity and encapsulation. *Theoretical Computer Science* 376, 3 (2007), 205–224.
[47] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *OOPSLA Companion*. 815–816.
[48] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 75–84.
[49] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.
[50] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.
[51] Esteban Pavese, Ezekiel O. Soremekun, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2018. Inputs from Hell: Generating Uncommon Inputs from Common Samples. *CoRR* abs/1812.07525 (2018). arXiv:1812.07525 http://arxiv.org/abs/1812.07525
[52] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. 2004. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering* 16, 11 (2004), 1424–1440.
[53] Doron Peled, Moshe Y Vardi, and Mihalis Yannakakis. 1999. Black box checking. In *Formal Methods for Protocol Engineering and Distributed Systems*. Springer, 225–240.
[54] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (Testing Tools Track)*.
[55] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*. IEEE, 46–57.
[56] Arjun Radhakrishna, Nicholas V Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Cernỳ. 2018. DroidStar: callback typestates for Android classes. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1160–1170.
[57] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*. Springer, 93–108.
[58] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366–401.
[59] Hossein Safyallah and Kamran Sartipi. 2006. Dynamic analysis of software systems using execution pattern mining. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 84–88.
[60] Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2016. Diversity-aware mutation adequacy criterion for improving fault detection capability. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 122–131.
[61] Peng Sun, Chris Brown, Ivan Beschastnikh, and Kathryn T Stolee. 2019. Mining Specifications from Documentation using a Crowd. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 275–286.
[62] Suresh Thummalapenta and Tao Xie. 2009. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 496–506.
[63] MP Vasilevskii. 1973. Failure diagnosis of automata. *Cybernetics* 9, 4 (1973), 653–665.
[64] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

[65] Tao Xie and David Notkin. 2003. Mutually enhancing test generation and specification inference. In *International Workshop on Formal Approaches to Software Testing*. Springer, 60–69.

[66] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*. ACM, 282–291.

[67] Hao Zhong and Zhendong Su. 2013. Detecting API documentation errors. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 803–816.