SYSMODIS: A Systematic Model Discovery Approach

Omer Korkmaz  
Faculty of Engineering and Natural Sciences  
Sabanci University  
Istanbul, Turkey  
omerkorkmaz@alumni.sabanciuniv.edu

Cemal Yilmaz  
Faculty of Engineering and Natural Sciences  
Sabanci University  
Istanbul, Turkey  
cyilmaz@sabanciuniv.edu

Abstract—In this paper, we present an automated model discovery approach, called SYSMODIS, which uses covering arrays to systematically sample the input spaces. SYSMODIS discovers finite state machine-based models, where states represent distinct screens and the edges between the states represent the transitions between the screens. SYSMODIS also discovers the likely guard conditions for the transitions, i.e., the conditions that must be satisfied before the transitions can be taken. For the first time a previously unseen screen is visited, a covering array-based test suite for the input fields present on the screen as well as the actions that can be taken on the screen, is created. SYSMODIS keeps on crawling until all the test suites for all the screens have been exhaustively tested. Once the crawling is over, the results of the test suites are fed to a machine learning algorithm on a per screen basis to determine the likely guard conditions. In the experiments we carried out to evaluate the proposed approach, we observed that SYSMODIS profoundly improved the state/screen coverage, transition coverage, and/or the accuracy of the predicted guard conditions, compared to the existing approaches studied in the paper.

Index Terms—Automated model discovery; systematic sampling; covering arrays; combinatorial testing

I. INTRODUCTION

Mobile devices have been increasingly becoming smarter and more powerful. Therefore, mobile applications in many areas such as education, health, economy, or management are used by millions of people on a daily basis. As the failures in the field may have some severe consequences, these applications need to be tested thoroughly.

One frequently used approach for this purpose is model-based testing [1]–[5]. In model-based testing, given a model representing the behavior of the system under test (SUT), test cases are automatically generated typically by employing a structural coverage criterion, such as those based on state and transition coverage [6], [7]. Many empirical studies strongly suggest that model-based testing is an efficient and effective approach for testing mobile applications [1]–[5]. One downside of model-based testing, however, is that it takes as input the model of the SUT. As these models often need to be created and maintained manually, the practicality of the model-based approaches, which, otherwise, can be quite effective, is hindered.

Many approaches have been proposed in the past to automatically discover the models of software systems, especially the mobile applications, so that the discovered models can be used with various quality assurance (QA) activities, including model-based testing [8]–[14]. One observation we make about the existing approaches for model discovery is that they often do not systematically take the interactions between system’s inputs into account.

In this paper, we conjecture that systematically sampling the input spaces by taking the interactions between input parameters into account can greatly improve the effectiveness of model discovery. To this end, we present an automated model discovery approach. SYSMODIS discovers finite state machine-based models [15], where states represent distinct screens discovered during crawling and the edges between states depict the transitions between screens. The transitions are further annotated with guard conditions, which represent the conditions that must be satisfied before the transitions can be taken.

To systematically, sample the input spaces, SYSMODIS uses a well-known combinatorial object for testing, called t-way covering arrays [16]. A t-way covering array, where $t$ is often referred to as the coverage strength, takes as input an input space model. In its simplest form, the model includes a set of parameters, each of which takes its value from a discrete domain. Given a model, a t-way covering array is a set of rows (where each row is comprised of values for all the parameters in the model), in which each possible combination of parameter values for every combination of $t$ parameters appears at least once [17], [18]. The basic justification for using t-way covering arrays is that they (under certain assumptions) can efficiently and effectively exercise all program behaviors caused by the interaction of $t$ or fewer parameters [16].

Covering arrays have been extensively used for software testing [10], [11], [19], [20]. In this work, however, we use them (and, to the best of our knowledge, for the first time) to systematically sample the input spaces for automated model discovery.

At a very high level, the proposed approach operates as follows: Starting with an initially empty model, every time, a previously unseen screen is encountered during crawling, a new state (together with the respective transition) is added into the model. The screen (i.e., the state) is then associated with
a covering array created for testing the interactions between the interactable user interface (UI) elements on the screen. For each screen, SYSMODIS aims to run all the test cases associated with the screen. More specifically, every time a screen is visited, a previously untested test case (otherwise; a randomly selected test case) associated with the screen is executed. After executing the test case, an appropriate state and/or transition is added into the model. The crawling process terminates when all the test cases for each discovered screen have been executed. After the crawling process, the results of the test cases are fed to a machine learning algorithm to identify likely guard conditions for the transitions on a per screen basis.

Note that many of the details of the proposed approach, such as the way the models are represented and the distinct states are determined as well as the opportunistic crawling strategy employed by the approach can readily be replaced by other strategies. As our ultimate goal in this study is to study the effectiveness of taking the interactions between input parameters into account for model discovery, we opted to use well-known and relatively easy-to-implement (from the perspective of correctness) approaches to resolve these technical challenges.

To evaluate the proposed approach, we have conducted a series of experiments. In the first set of experiments, we carried out controlled experiments where we systematically varied the model and process parameters to study the sensitivity of the proposed approach to these parameters. In the second set of experiments, we carried out comparative studies on real subject applications where we compared the results obtained from the proposed approach to those obtained from some existing approaches [8], [9] as well as from random testing. We observed that the proposed approach profoundly improved the state/screen coverage, transition coverage, and/or the accuracy of the predicted guard conditions, compared to the existing approaches used in the paper.

The remainder of the paper is organized as follows: Section II introduces the proposed approach; Section III presents the experiments carried out to evaluate the proposed approach; Section IV discusses threats to validity; Section V discusses related work; and Section VI presents concluding remarks and discusses possible future works.

II. APPROACH

SYSMODIS discovers a finite state machine-based model for the subject application under test (SUT). In the model, while the states represent the distinct screens discovered during crawling, the edges represent the transitions between the screens. Each transition is further associated with a label and a guard condition, indicating the action that needs to be taken and the condition that needs to be satisfied before the transition can be taken. For example, when a user interacts with a login screen, if Login Name is valid, but Password is not, then the user will be directed to an invalid password screen. In this scenario, one of the guard conditions for the login screen is, therefore, a combination of valid Login Name with invalid Password.

At a very high level, SYSMODIS operates as depicted in Figure 1. Every time a screen is encountered during crawling, a check is performed to determine whether the screen has been seen before. If the screen is a previously unseen screen, then a new state is inserted into the model. For each newly discovered screen (i.e., newly added state), the screen is first analyzed and the input elements (e.g., Login Name) – user interface (UI) elements, which expect to receive some input from the end user, as well as the actions (e.g., clicking on the Login button), which can be taken by the end user, are determined. Then, a standard covering array for the input elements is created and associated with the screen (i.e., with the respective state). To this end, each input element on a screen is first associated with an input domain, for which a pre-determined set of equivalence classes [21] is given. Then, a covering array for the screen is created by expressing the input elements as parameters, the values of which are drawn from the respective equivalence classes.

Each row in the covering array created for a screen represents a collection of input values that should be fed to the respective UI elements on the screen. For each screen, SYSMODIS aims to test all the selected combinations of input values by paring each combination with all possible actions that can be taken on the screen. In the remainder of the paper, a collection of input values for the UI elements present on a screen (one value for each UI element), e.g., Login Name = Jhon and Password = Doe, together with the action to be taken, e.g., click on the Login button, will be referred to as a test case.

To run a test case on a screen, the input values determined by the test case are fed to the respective UI elements and then the selected action is taken. As a result of running a test case, a transition (if not already present) is inserted into the model from the state, on which the test case was executed, to the state (i.e., screen), to which the system is moved, with a label indicating the action causing the transition.

The collection of all test cases created for a screen constitutes a test suite for the screen. Every time the screen is visited during crawling, a previously untested test case is selected from its test suite and run. When the test suite have already been executed exhaustively, a randomly selected test case from the suite is run. The crawling process terminates once all the test suites created for the screens are executed exhaustively.

After the crawling process, the results of the test suites (i.e., test cases and resulting states after they are run) are fed to a machine learning algorithm on a per state basis to determine the guard conditions for the transitions originating from the state. As the guard conditions are typically required to be interpreted by the users of the discovered model, e.g., software engineers, who use the model to reason about the system’s behavior or model-based testing approaches, which generate test cases based on the model, we opt to discover human-readable guard conditions.

Next we provide further details about the proposed ap-
Note that the ultimate goal of this work is to study the effectiveness of systematically sampling the input spaces for model discovery by using covering arrays. Therefore, many of the specific approaches and techniques used in SYMSMODIS, such as detecting distinct screens and determining the domain of input elements, concern the tools we used to resolve some technical issues, rather than the contribution of this work. We, therefore, opted to use relatively easy-to-implement (from the perspective of functional correctness), yet well-known approaches to resolve these technical issues, rather than attempting to find optimal solutions to them, and implemented SYMSMODIS on a well-known computing platform, namely Android. Note, however, that the proposed approach can work with alternative approaches for resolving the aforementioned technical issues and is readily applicable to other platforms, including mobile platforms, such as iOS, or web applications.

A. Capturing Screens

To analyze a screen encountered during crawling, we first capture the screen in the form of an XML document. This document defines the logical structure of the UI elements present on the screen together with some metadata information about these elements, such as their types (e.g., buttons or text fields) and their attributes, especially the ones indicating the way that they can be interacted with (e.g., clickable and scrollable).

B. Determining UI Elements

We then analyze the screen capture (by systematically traversing the XML document) to detect the interactable UI elements, with which the end user can interact. In particular, we distinguish between two types of interactable elements; actionable elements and input elements. While the actionable elements are the UI elements that are clickable, double clickable, or scrollable, the input elements are all the remaining interactable UI elements that await input from the user, such as, text fields and spinners.

C. Determining Distinct Screens

To determine whether a screen has been previously seen (i.e., whether the screen can be mapped to a state in the model), we hash the screen to a value. If the hash value is associated with an existing state in the model, then the screen has been seen before. Otherwise, a new screen has been encountered.

To this end, we use the attributes of the interactable UI elements, including their class names and resource IDs, in an order-agnostic manner (i.e., independent of the order, in which these elements are processed) by simply ordering the elements according to their attributes. The resulting data is then fed to the MD5 hashing function [22].

D. Determining Input Domains and Equivalence Classes

Once a screen is discovered for the first time, we generate a standard covering array to systematically sample the input space for the screen by treating each input element present on the screen as a parameter.

To determine the values that can be assumed by a parameter (i.e., the input values that can be fed to the respective UI element), we first determine the input domain, then use this information to locate the pre-determined equivalence classes given for this domain, next draw one input value from each equivalence class, and finally use all the input values drawn as the set of values that can be assumed by the parameter.

For example, the default equivalence classes we define for the age domain are: newborns (less than 2 months old), infants (between 2 months and 1 year old), toddlers (1-4 years old), children (5-11 years old), young teenagers (12-14 years old), teenagers (15-17 years old), young adults (18-35 years old), middle-aged adults (36–55 years old), and older adults (more than 55 years old). That is, when an input element requiring input from this domain is encountered, a new parameter with 9 possible values (one randomly drawn value per equivalence class) is created. SYMSMODIS is designed, such that the repository of domains can be updated by adding new domains and/or by modifying/removing existing domains. For example, if the equivalence class definitions given above are
not appropriate for the subject application under test, then they can be modified.

To determine the domain of an input element, as this is out of the scope of this work, we use a simple, keyword-based approach. More specifically, each domain definition in SYSMODIS is associated with a modifiable list of keywords, describing the domain. If the certain attributes of an input element, i.e., the label or developer/user hints (if present), contains one of these keywords, the element is labeled with the respective domain. SYSMODIS verifies its repository to make sure that keyword lists are non-overlapping.

E. Generating Covering Arrays

For a newly discovered screen, once the parameters and their possible values are determined, a \( t \)-way covering array is generated. In this array, each row corresponds to a collection of input values to be fed to the input elements. Each row is then converted to (possibly) multiple test cases by pairing the row with every possible action that can be taken on actionable elements present on the screen. The collection of the test cases created for a screen constitutes the test suite for the screen.

F. Crawling

SYSMODIS employs an opportunistic crawling strategy. When a screen is visited, a previously untested test case is selected form the associated test suite and executed. If the test suite has already been exhaustively executed, then a test case, which can take the system towards the nearest state with some untested test cases, is executed. This process is repeated for each state encountered during crawling. Furthermore, SYSMODIS can be configured to restart the subject application under test after executing a pre-determined number of test cases – a precaution we take to prevent the crawling process from getting stuck as much as possible.

G. Discovering Likely Guard Conditions

After the crawling process has been terminated, the results of the test cases are analyzed on a per-state basis to determine likely guard conditions for the transitions. The guard condition for a transition from state \( S \) to state \( S' \) is a condition defined over the input elements present on \( S \), indicating the condition (i.e., combination of input values required) in order the transition to be taken. As the guard conditions need to be interpreted either manually by software engineers or automatically by other model-based approaches, we opted to discover guard conditions, which can both be readable by humans and be processed by automated approaches.

To this end, we cast the problem to a classification problem and use a decision tree classifier to identify likely guard conditions [23].

More specifically, for each state \( S \) in the model, every test case that was executed represents a record where the input elements and the action used to run the test constitute the attributes of the record and the state, to which the system is moved after executing the test, constitutes the label for the record. In the remainder of the paper, the collection of these records will be referred to as a training set.

Inspired by [23], we train a single classification tree model for each state \( S' \), which \( S \) is adjacent to. More specifically, for each distinct adjacent state \( S' \) (i.e., for each distinct label in the training set), we create a separate data set by keeping all the records with the label \( S' \) as they are and replacing the labels of the remaining records with \( \text{non} \) – \( S' \). The constructed data set is then fed to a decision tree classifier. The output is a binary classification tree where the leaf nodes are marked with either \( S' \) or \( \text{non} \) – \( S' \) and the edges are labeled with a condition over the value of a single input element. For every leaf node marked with \( S' \), the conjunction of the edge conditions on the path from the root to the leaf node constitute a condition from moving \( S \) to \( S' \). Consequently, the likely guard condition for the transition from state \( S \) to state \( S' \), is expressed as the disjunction of all of these conditions (one per leaf marked with \( S' \)).

III. EXPERIMENTS

We have conducted a series of experiments to evaluate the proposed approach. In the first set of experiments (Section III-A), we have evaluated the sensitivity of the approach to various model and process parameters. To this end, we have used simulations as it was not possible for us to systematically vary these parameters on real applications, on which we had no control over. In the second set of experiments (Section III-B), we have evaluated the proposed approach by conducting comparative studies using real subject applications.

A. Evaluating Sensitivity to Model Parameters

In this set of experiments, we evaluate the sensitivity of the proposed approach to the model parameters by systematically varying these parameters via simulations.

Setup. In particular, we manipulate the parameters given in Table I:

- states: The number of states in the model.
- density: The graph density of the model [24], which determines the number of transitions in the model.
- parameters: The number of parameters defined for a state, i.e., the number of input elements on the respective screen.
- settings: The number of equivalence classes for a parameter.
- guard-complexity: The number of equivalence classes for a parameter.
- non-determinism: The level of non-determinism, depicting the probability of not taking the transition that needs to be taken, which mimics the non-deterministic situations in real life. When non-determinism \( = 0 \), the model is deterministic.

Table I presents the values used for these parameters in the experiments, which were determined based on our experience in testing. If the value of a parameter indicates a range (as is the case in the number of parameters per state and the number of equivalence classes per parameter), the actual value to be
used is randomly chosen from the given range. Furthermore, guard conditions are expressed in the form of a conjunction of the input element and value pairs, in a way that guarantees that the guard conditions of the transitions originating from a state are pairwise mutually exclusive. Note the the case of non-determinism is studied with the help of the non-determinism parameter described above.

To carry out the study, we randomly created 100 models for each configuration in the Cartesian product of the independent variables given in Table I.

**Evaluation Framework.** To evaluate the proposed approach on these models, we use the following metrics:

- **state coverage:** Percentage of the states visited (i.e., discovered).
- **transition coverage:** Percentage of the transitions traversed.
- **accuracy:** Accuracy of the guard conditions predicted.

The higher the state coverage, transition coverage, and accuracy, the better the proposed approach is. When computing the accuracy, given a transition $E$ from state $S$ to state $S'$ with a guard condition $C$ in the model, if the predicted condition $C'$ deterministically takes the system from $S$ to $S'$, i.e., if $C'$ is not satisfiable with any of the guard conditions associated with the transitions originating from $S$, except for $E$, then the prediction is counted as true positive. The accuracy is then computed as the percentage of the true positives.

We, furthermore, compare the results obtained from the proposed approach to those obtained from random testing. To this end, we used the same models. For each state that the random testing strategy visited during crawling, we randomly generated the same number of test cases by using the same equivalence classes with the proposed approach. Everything else in the experiments was kept the same, i.e., the way the models were simulated and the way the results were evaluated.

**Operational Framework.** All the experiments were carried out on an Intel I7 6700HQ machine with 16 GB of RAM, running Windows 10. The covering arrays were generated by using ACTS [25] and the classification tree models were trained by using Decision Tree Classifier implemented in scikit-learn [26].

**Data and Analysis.** In total, we generated 22,997,103 test cases and trained 7,200 classification models by using the proposed approach. The results we obtained are summarized in Tables II, III, IV and Figures 2, 3, 4.

We first observed that as the strength of the covering arrays used for model discovery increased, state coverage, transition coverage, and accuracy of the predicted guard conditions all increased (Table II). This is, indeed, to be expected as higher strength covering arrays cover more distinct combinations of parameter values. Overall, while the average state coverage, transition coverage, and accuracy obtained from 2-way coverings were 83.10%, 82.28%, and 68.07%, those obtained from 3-way covering arrays were 91.90%, 91.48%, and 73.12%, respectively. And, for 4-way covering arrays, we obtained 100% state and transition coverage, and 77.20% accuracy.

We then focused on the deterministic models (where non-determinism = 0). When $t$ (i.e., the coverage strength) was more than or equal to guard-complexity in the deterministic experiments, SYSMODIS obtained perfect (100%) state and transition coverage regardless of the model parameters used (Figures 2-3). This was because $t$-way covering arrays guaranteed to have all possible combinations of the parameter values used in the guard conditions, which, in turn, guarantees to have at least one test case satisfying each guard condition. When $t$ was smaller than guard-complexity, state and transition coverage tended to increase as $t$ increased (Figures 2-3). For example, when guard-complexity $= 4$, the average state and transition coverage values were 56.45% and 55.05% for $t = 2$ and 67.60% and 65.93% for $t = 3$.

Regarding the accuracy of the predicted guard conditions, we observed similar trends. More specifically, for a fixed guard-complexity, the accuracy tended to increase as $t$ increased (Figure 4). For example, when the guard-complexity $= 2$, the average accuracies were 73.71%, 78.45%, and 80.75% for $t = 2$, 3, and 4, respectively. Furthermore, as guard-complexities increased, the accuracies tended to decrease (Figure 4). For example, when $t = 4$, the average accuracies were 84.50%, 80.75%, 77.19%, and 73.62% for guard-complexity $= 1, 2, 3,$ and $4,$ respectively.

Based on these results, we suggest using SYSMODIS (if feasible) with a coverage strength, which is greater than the maximum number of parameters that may be involved in guard conditions. If this is not feasible, then the largest affordable coverage strength should be used. Regarding the effect of non-determinism, we expectedly observed that as the level of non-determinism increased, the state coverage, transition coverage, and accuracy values

---

**Table I**

| Variable | Values |
|----------|--------|
| Number of states | $\{10, 20, 50\}$ |
| Model density | $\{0.4, 0.6\}$ |
| Number of parameters per state | $\{5, 10, 16, 20\}$ |
| Number of equivalence classes per parameter | $\{3, 6\}$ |
| Number of distinct parameters in guard conditions | $\{1, 2, 3, 4\}$ |
| Coverage strength | $\{2, 3, 4\}$ |
| Level of non-determinism | $\{0, 0.01, 0.05, 0.1\}$ |

**Table II**

| $t$ | Screen Coverage | Transition Coverage | Accuracy |
|-----|-----------------|---------------------|----------|
| 2   | 83.10%          | 82.28%              | 68.07%   |
| 3   | 91.90%          | 91.48%              | 73.12%   |
| 4   | 100.00%         | 100.00%             | 77.20%   |
tended to decrease. Table III presents the results we obtained from the experimental setups where $t$ was greater than guard-complexity. When the level of non-determinism was at the highest level used in the experiments (i.e., when non-determinism = 0.10), we, on average, obtained 93.56% statement coverage, 92.23% transition coverage, and 68.81% accuracy.

Last but not least, by comparing the results obtained from SYSMODIS to those obtained from random testing, we observed that the proposed approach performed significantly better than random testing. For example, when non-determinism = 0 and $t \geq$ guard-complexity, while the average state coverage, transition coverage, and accuracy obtained from random testing were 71.34%, 70.89%, and 73.78%, respectively, those obtained from SYSMODIS were 100.00%, 100.00%, and 81.89%.

### B. Evaluations on Real Subject Applications

In this section, we evaluate the proposed approach by conducting comparative studies using real subject applications. In particular, we compare the results obtained from the proposed approach to those obtained from Monkey [8], Dynodroid [9], and random testing.

**Setup.** In the experiments, we used 10 subject applications from Google Play Store [27]. Table V presents information about these subjects. We chose these applications because they had also been used to evaluate related approaches in the literature [2], [4], [9], [11]. Furthermore, to determine the input domains and their equivalence classes, we used our knowledge of the domain of the application.

**Evaluation Framework.** For the evaluations, as the true models of the subject applications were not known, we were not able to compute the state coverage, transition coverage, and the accuracy metrics we used in the previous study (Section III-A). Instead, we used the code coverage and screen coverage metrics, which measure the percentage of the source code statements executed and the percentage of the screens discovered during crawling, respectively. In this context, a screen corresponds either to an Android activity or to a window, such as a pop-up or a modal window. Given a subject application, we determined the number of distinct screens (as described in Section II-C) by first performing a static analysis.

---

**TABLE III**

**EFFECT OF NON-DETERMINISM WHEN $t >$ guard – condition.**

| Non-determinism | State Cov. | Transition Cov. | Accuracy |
|-----------------|------------|-----------------|----------|
| 0.00            | 100.00%    | 100.00%         | 81.89%   |
| 0.01            | 98.42%     | 98.51%          | 76.02%   |
| 0.05            | 96.04%     | 96.07%          | 72.63%   |
| 0.10            | 93.56%     | 92.23%          | 68.81%   |
of the application. Then, as we discovered new screens during crawling (either by the proposed approach or by the alternative approaches) we updated the set of known screens (Table V).

For the comparisons, we made sure that all the approaches carried out the same number of test steps. To this end, feeding a value to an input element or taking an action is counted as a single test step. We couldn’t directly compare the actual number of test cases executed (as defined in Section II-E), since some of the alternative approaches, namely Monkey and Dynodroid, did not have a notion of test cases. Similarly, as Monkey and Dynodroid did not have a notion of “screens,” we couldn’t collect the screen coverage statistics for them.

**Operational Framework.** In the experiments, we used 1) ACTS [25] to generate the covering arrays, 2) Appium [28] to execute the generated test cases on the subject applications, 3) ACVTool [29] to measure the code coverage obtained without needing to have the source codes of the applications, and 4) Decision Tree Classifier in scikit-learn [26] to predict the guard conditions. All the experiments were carried out on the same computing platform we used in Section III-A.

**Data and Analysis.** Tables VI-VIII presents the results we obtained. Overall, we first observed that SYSMODIS provided an average of 93.37% screen coverage and up to 65.37% code coverage (Table VI). We then observed that it performed profoundly better than random testing; random testing provided up to 83.21% screen coverage and up to 58.35% code coverage (Table VI).

Table VII presents the itemized results. In this table, for each coverage strength used in the experiments, the columns indicate the total number of rows included in the covering arrays computed for the study, the total number of test cases created (Section II-E), the total number of test steps carried out, the total execution time in minutes, and the screen coverage respectively (code coverage statistics can be found in Table VIII). As the coverage strength of the covering arrays increased, the screen coverage statistics improved for all the subject applications. The average, screen coverage percentages were 92.14%, 93.49%, and 94.58% for $t = 2$, 3, and 4, respectively (Table VII).

Similarly, Table VIII presents the code coverage statistics obtained. As the coverage strength increased, the code coverage statistics increased. And, the proposed approach consistently performed better than the alternative approaches. The code coverage statistics obtained from SYSMODIS were 61.40%, 64.10%, and 66.80%, on average, when $t = 2$, 3, and 4, whereas the coverage percentages were 52.20%, 54.50%, and 56.50% for random testing, 38.30%, 40.90%, and 43.50% for Monkey, and 39.66%, 41.88%, and 43.88% for Dynodroid.

Interestingly enough, the random testing strategy employed in the experiments performed better than Monkey and Dynodroid, demonstrating the relevance and importance of using equivalence classes for model discovery. And, systematically covering the interactions between them, as was the case with SYSMODIS, further improved the results.

### IV. Threats to Validity

One external threat concerns the representativeness of the subject applications used in the study. To alleviate this issue, we used 10 subject applications, which had also been used for evaluating related approaches in the literature [4], [8], [9], [11], [30]. We, furthermore, carried out simulations, where we systematically varied the model and process parameters.

Another threat concerns the representativeness of the computing platform used in the study. To address this issue we used a well-known and frequently-used platform, namely Android. After all, all the requirements that the proposed approach expects the underlying platform to satisfy (such as, accessing the information of the UI elements on the screen and a means of automatically interacting with these elements) are also met by other platforms, such as iOS and web platforms.

SYSMODIS takes equivalence classes to be used as input. To this end, it provides an extensible mechanism (by using the adapter design pattern), which allows new input domains and equivalence classes easily to be plugged in. Note that once provided, this information may not need to change during the lifetime of the application under test, unless new input types are introduced in the application. Note, further, that applications from the same domain, as they process the same types of inputs, can share the same/similar repository of

---

**TABLE IV**

| Approach     | State Coverage | Transition Coverage | Accuracy   |
|--------------|----------------|---------------------|------------|
| SYSMODIS     | 100.00%        | 100.00%             | 81.89%     |
| Random       | 71.34%         | 70.89%              | 73.78%     |

**TABLE V**

| SUT          | Description            | Total Screens |
|--------------|------------------------|---------------|
| Tureng      | Word translator        | 11            |
| Tippy Tipper| Tip calculator         | 18            |
| Munchlife   | Counter/bookkeeper     | 5             |
| Contact Manager| Contact manager     | 2             |
| To Do Manager| To-do manager         | 7             |
| Alarm Klock | Alarm clock            | 8             |
| Habit App   | Habit tracker          | 3             |
| Any Memo    | Foreign language learning aid | 40 |
| Weight Tracker| Weight tracker      | 15            |
| BBC News    | News channel           | 10            |

**TABLE VI**

| t  | Approach | Screen Coverage | Code Coverage |
|----|----------|-----------------|---------------|
| 2  | SYSMODIS | 93.37%          | 56.02%        |
| 3  | SYSMODIS | 93.37%          | 60.86%        |
| 4  | SYSMODIS | 93.37%          | 65.37%        |
| 2  | Random   | 79.37%          | 44.02%        |
| 3  | Random   | 81.33%          | 56.86%        |
| 4  | Random   | 83.21%          | 58.35%        |
equivalence classes. Therefore, a repository that is created for one application can also be used for another application in the same domain.

V. RELATED WORK

Existing approaches in the literature can broadly be grouped into three categories: random testing-based, model-based, and goal oriented [1]–[4], [8]–[11], [13], [31], [32].

Random testing-based approaches crawl/test the application under test typically by generating random UI/system events and random inputs [8], [9], [12], [33]. Compared to random testing-based approaches, SYSMODIS uses an equivalence classes-based testing approach.

Some model-based approaches generate test cases based on the model of the application, which is either provided manually [34] or discovered by using static analysis of the software artifacts [4], [5], [30]. Some of the latter approaches even use covering arrays for generating test cases [5]. Compared to the former set of approaches, SYSMODIS automatically discovers the models. Compared to the latter set of experiments, SYSMODIS discovers the models at runtime by carrying out dynamic analyses. That is, SYSMODIS uses covering arrays to discover the models, not to test the applications given their models.

Other model-based approaches also aim to dynamically discover the models [1]–[3]. These approaches, however, do not systematically test the interactions between different input parameters, which SYSMODIS achieves by using covering arrays. In goal-oriented approaches, the application is crawled and/or tested in a goal-oriented manner [10], [11], [13]. For example, Evodroid [11] uses evolutionary algorithms to produce inputs, such that the code coverage obtained is maximized as much as possible. Similarly, ACTEve [10] uses a concolic-testing tool to improve path coverage. Even in these approaches, however, systematic testing of the input parameters is generally overlooked.

VI. CONCLUDING REMARKS

In this work, we have studied the effectiveness of systematically sampling the input spaces for model discovery by using covering arrays. To this end, we have developed an approach and evaluated it both by carrying out controlled experiments where we systematically varied the model and the process parameters to study the sensitivity of the proposed approach.

### TABLE VII

| SUT            | t = 2 | t = 3 | t = 4 |
|----------------|-------|-------|-------|
|                | CA Size | Test Cases | Test Steps | Exec. Time | Screen Cov. | CA Size | Test Cases | Test Steps | Exec. Time | Screen Cov. | CA Size | Test Cases | Test Steps | Exec. Time | Screen Cov. |
| Tureng         | 70     | 185    | 297    | 72%       | 520       | 55%     | 97       | 182      | 35%       | 65%       |
| To Do Manager  | 33     | 97     | 182    | 100%      | 34 min    | 100.00% | 126      | 150      | 35 min    | 100.00%   | 267      | 300      | 54 min    | 84.88%    |
| Tippy Tipper   | 21     | 75     | 121    | 100%      | 34 min    | 100.00% | 90       | 176      | 310 min   | 100.00%   | 268      | 297      | 56 min    | 85.33%    |
| Munchlife      | 9      | 19     | 125    | 100%      | 34 min    | 100.00% | 25       | 194      | 389 min   | 100.00%   | 90       | 194      | 36 min    | 76.66%    |
| Alarm Klock    | 28     | 79     | 160    | 100%      | 34 min    | 100.00% | 87       | 194      | 389 min   | 100.00%   | 128      | 210      | 52 min    | 65.33%    |
| Habit App      | 34     | 90     | 176    | 100%      | 34 min    | 100.00% | 102      | 194      | 389 min   | 100.00%   | 268      | 297      | 56 min    | 76.66%    |
| Weight Tracker | 25     | 67     | 136    | 100%      | 34 min    | 100.00% | 78       | 194      | 389 min   | 100.00%   | 128      | 210      | 52 min    | 65.33%    |
| BBC News       | 76     | 151    | 90     | 100%      | 34 min    | 100.00% | 183      | 194      | 389 min   | 100.00%   | 268      | 297      | 56 min    | 76.66%    |
| Any Memo       | 30     | 92     | 179    | 100%      | 34 min    | 100.00% | 101      | 194      | 389 min   | 100.00%   | 268      | 297      | 56 min    | 76.66%    |
| Contact Manager| 37     | 125    | 222    | 100%      | 34 min    | 100.00% | 148      | 267      | 520 min   | 100.00%   | 222      | 267      | 55 min    | 100.00%   |

### TABLE VIII

| SUT            | t = 2 | t = 3 | t = 4 |
|----------------|-------|-------|-------|
|                | CA Size | Test Cases | Test Steps | Exec. Time | Screen Cov. | CA Size | Test Cases | Test Steps | Exec. Time | Screen Cov. | CA Size | Test Cases | Test Steps | Exec. Time | Screen Cov. |
| Tureng         | 51%    | 54%    | 58%    | 43%       | 47%       | 49%     | 29%       | 34%       | 38%       | 42%       | 33%     | 34%       | 42%       | 44%       |
| To Do Manager  | 64%    | 66%    | 68%    | 56%       | 59%       | 60%     | 36%       | 39%       | 41%       | 44%       | 41%     | 41%       | 47%       | 47%       |
| Tippy Tipper   | 76%    | 79%    | 81%    | 62%       | 64%       | 67%     | 45%       | 49%       | 52%       | 53%       | 53%     | 53%       | 53%       | 53%       |
| Munchlife      | 74%    | 76%    | 69%    | 63%       | 65%       | 69%     | 42%       | 44%       | 59%       | 58%       | 59%     | 58%       | 58%       | 58%       |
| Alarm Klock    | 63%    | 65%    | 66%    | 55%       | 57%       | 60%     | 40%       | 43%       | 46%       | 46%       | 46%     | 46%       | 46%       | 46%       |
| Habit App      | 57%    | 62%    | 65%    | 46%       | 48%       | 60%     | 36%       | 38%       | 48%       | 48%       | 48%     | 48%       | 48%       | 48%       |
| Weight Tracker | 63%    | 65%    | 66%    | 57%       | 59%       | 60%     | 44%       | 47%       | 52%       | 52%       | 52%     | 52%       | 52%       | 52%       |
| BBC News       | 49%    | 52%    | 53%    | 41%       | 43%       | 44%     | 26%       | 28%       | 32%       | 32%       | 32%     | 32%       | 32%       | 32%       |
| Any Memo       | 50%    | 52%    | 56%    | 42%       | 44%       | 45%     | 36%       | 37%       | 40%       | 40%       | 40%     | 40%       | 40%       | 40%       |
| Contact Manager| 67%    | 70%    | 72%    | 57%       | 59%       | 63%     | 49%       | 50%       | 52%       | 52%       | 52%     | 52%       | 52%       | 52%       |
to these parameters and by conducting comparative studies on real subject applications.

Note that many of the details of the proposed approach, such as the way the model is represented and the distinct states are determined as well as the opportunistic crawling strategy employed by the approach can readily be replaced by other strategies. However, the results of our experiments strongly support our hypothesis that taking the interaction between input parameters into account can improve model discovery. We have arrived at this conclusion by noting that compared to alternative strategies, the proposed approach profoundly improved the state/screen coverage, transition coverage, and/or the accuracy of the predicted guard conditions.

One possible avenue for future research is to develop feedback-driven crawling strategies, which take into account not only the interactions within a state but also the interactions across the states. Another avenue is to evaluate the proposed approach using additional subject applications running on different platforms.

ACKNOWLEDGMENTS

This research was supported by the Scientific and Technological Research Council of Turkey (118E204).

REFERENCES

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, page 258–261, 2012.
[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: a tool for automated model-based testing of mobile apps. IEEE Software, 32(5):53–59, 2015.
[3] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys ’14, page 204–217, 2014.
[4] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE’13, page 250–265, 2013.
[5] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in gui testing of android applications. International Conference on Software Engineering,ICSE’16, 2016.
[6] Shafique M. and Labiche Y. A systematic review of model based testing tool support (technical report sce-10-04), carleton university, canada. 2010.
[7] Pradhans S., Ray M., and S. Patnaik. Coverage criteria for state-based testing: A systematic review. International Journal of Information Technology Project Management (IJITPM), 2019.
[8] Monkey. https://developer.android.com/studio/test/monkey. 2018.
[9] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, pages 224–234, 2013.
[10] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE ’12, 59:1–11, 2012.
[11] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE’14, 2014.
[12] Claessen Koen and Hughes John. Quickcheck: A lightweight tool for random testing of haskell programs. SIGPLAN Not., 35(9):268–279, September 2000.
[13] H. van der Merwe, B. van der Merwe, and W. Visser. Execution and property specifications for jpt-android. SIGSOFT Software Engineering Notes, pages 1–5, 2014.
[14] Michael Benedikt, Juliana Freire, and Patrice Godefroid. Veriweb: Automatically testing dynamic web sites. In Proceedings of 11th International World Wide Web Conference, 2002.
[15] A.Andrews, J.Offutt, and R.T. Alexander. Testing web applications by modeling with fsms. Software & Systems Modeling, pages 326–345, 2005.
[16] D. M. Cohen, S. R. Dalal, M. L. Fredman, , and G. C. Patton. The aetg system: an approach to testing based on combinatorial design. IEEE Trans. on Soft. Eng., 23(7):437–444, 1997.
[17] C. Yilmaz, S. Fouche, M. B. Cohen, A. Porter, G. Demiroz, and U. Koc. Moving forward with combinatorial interaction testing. Computer, (2):37–45, 2014.
[18] C. Nie and H. Leung. A survey of combinatorial testing. ACM Computing Surveys (CSUR), 43(2), 2011.
[19] S. Manchester. Combinatorial-based prioritization for user-session-based test suites. 2021.
[20] S. Sampath, R.C. Bryce, S. Jain, and S. Manchester. A tool for combination-based prioritization and reduction of user-session-based test suites. 27th IEEE International Conference on Software Maintenance (ICSM), pages 574–577, 2011.
[21] A. Bhat and S. M. K. Quadri. Equivalence class partitioning and boundary value analysis - a review. pages 1557–1562, 2015.
[22] Ronald L. Rivest. The MD5 Message-Digest Algorithm. (1321), April 1992.
[23] Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Covering arrays for efficient fault characterization in complex configuration spaces. SIGSOFT Softw. Eng. Notes, 29(4):45–54, July 2004.
[24] Ravindra K. Ahuja. Network flows: Theory, algorithms, and applications (1st ed.). Pearson Education., 2017.
[25] ACTS. Automated combinatorial testing for software.
https://csrc.nist.gov/Projects/automated-combinatorial-testing-for-software., 2018.
[26] Scikit-learn. https://scikit-learn.org/stable/, 2018.
[27] Google Play Store. https://play.google.com/store/, 2018.
[28] Appium. http://appium.io/, 2018.
[29] AcvTool. https://github.com/pilgun/acvtool., 2018.
[30] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? Proc. of ASE’15, pages 429–440, 2015.
[31] Z. Liu, X. Gao, and X.Long. Adaptive random testing of mobile application. International Conference on Computer Engineering and Technology, ICSET, pages 297–301, 2010.
[32] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro. Autoblacktest: Automatic black-box testing of interactive applications. International Conference on Software Testing, Verification and Validation, ICST’12, pages 81–90, 2012.
[33] C. Hu and I. Neamtiu. Automating gui testing for android applications. International Workshop on Automation of Software Test, AST’11, pages 77–83, 2011.
[34] Takala T., M. Katara, and J Harty. Experiences of system-level model-based gui testing of an android application. Proc. 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST’11, pages 377–386, 2011.