Leveraging Data Invariants in Model Inference for Test Case Generation

Roykrong Sukkerd

Abstract—Testing is an effective mean to find bugs in systems, but manually writing test cases is often tedious. Automated test generation techniques allow developers and maintainers to check many properties of their systems at reasonable cost. One important goal in test generation is to induce new system under test (SUT) behaviors. However, such new behaviors in SUT can also be derived with model inference techniques. The goal of model inference is to infer description of systems. Synoptic is a model inference tool that is especially suitable to test generation because it is generative and scalable. Synoptic infers concise and accurate models of systems from logs. Synoptic models have been used for comprehension. The fundamental challenge of using Synoptic models for test generation is that they are too abstract. I propose to reduce the gap between abstract models and system implementations by integrating data properties into Synoptic models. In this work, I present Radish, an approach that combines data invariants mined with Daikon with Synoptic models. Radish uses data invariant satisfiability as a heuristic for eliminating invalid behaviors of systems from models, for improving accuracy. Radish is a promising technique that enables Synoptic for test generation that we hope to evaluate in future work.

I. INTRODUCTION

Testing is the method by which most software is verified today. Testing many behaviors of a system increases confidence, but manually writing test cases is often a tedious process. One important task in testing is selecting what behaviors of the system to test. Model-inference techniques can aid such task and can enable automated test case generation.

Model-inference techniques aim to produce a model — typically a finite state machine (FSM) — that accurately and concisely represent a subject system. The inferred model has the potential to be used for test case generation. We can select a set of system behaviors described in the model (in case of an FSM model, a system behavior is a path in the model), and convert them into test cases that exercise those behaviors in the system.

Many efficient model-inference techniques use dynamic analysis. They infer models from observed system executions. To use those model-inference techniques in test case generation, we first must have a set of test cases that exercise some behaviors of the system. Once we infer a model of the system, we want to use it to produce more test cases to exercise behaviors that have not been covered previously. Therefore, for the inferred model to be useful in testing, it must be generative — it must describe plausible behaviors of the system that have not yet been observed.

Synoptic is a model-inference algorithm that produces generative models. Synoptic infers a concise and accurate finite state machine model of a system from logs the system produces. Synoptic makes generalizations of the system — the model it generates may also describe system behaviors that were not observed in the logs. For accuracy, Synoptic controls how it makes generalizations by preserving three kinds of temporal invariants mined from the logs in the model it produces. Synoptic model is suitable for test case generation, because the behaviors it predicts are likely to be real behaviors of the system.

There are 2 main steps in generating test cases from Synoptic model:

1) Select abstract test cases from the model. An abstract test case is a sequence of events that the model accepts. In this work, a sequence of events is referred to as a behavior of a system.

2) Convert abstract test cases into executable test cases. We must find input and/or configuration values that, when fed to the system, exercise the behaviors that the abstract test cases describe.

Unfortunately, Synoptic model cannot be used for test case generation as is. The fundamental issue is that Synoptic model is too abstract to derive executable test cases. Consider an example of a simple authentication system, VerifyPin, that grants access to the user if and only if the user provides the correct pin within three tries; otherwise, the system blocks the user from getting access. Once granted access, the user can obtain access again under the same condition, i.e., the user is allowed three tries again. Synoptic model of VerifyPin is shown in Figure 1. This model is not a sufficient description of VerifyPin. It does not describe what input triggers the system to grant or deny access to a user. It also describes illegal sequences of invocations in VerifyPin, namely, a user can be denied access three or more consecutive times and still be granted access later. The reason is because Synoptic only reasons about temporality of system events and neglects the data values on which those events depend.

There are two key variables that affect event temporality in VerifyPin: the input pin and the counter of remaining tries the user is allowed in order to get access. We want a model of VerifyPin that preserves the following properties:

• the user is granted access if and only if the input pin is correct and the user has not exceeded the allowed three tries
• the user is denied access (but could still get access later on) if and only if the input pin is incorrect and the user has not exceeded the allowed three tries
• otherwise, the user is blocked from getting access

We can express these properties by extending Synoptic model with annotation on edges that capture the constraints on data
values. Figure 2 shows an extended finite state machine (EFSM) model that captures all key properties of VerifyPin.

VerifyPin example illustrates 2 issues of Synoptic model:

1) Synoptic model describes illegal sequences of invocations in the subject system, because it does not reason about event sequences dependencies on data values.
2) Some invocation sequences are legal in the system only under certain conditions, but Synoptic model accepts those invocation sequences regardless of the conditions.

These issues can be addressed by detecting the likely data invariants during the system executions and annotating them on the models edges as path constraints. Once the model has data constraints on its edges, the model describes the subject system more accurately, that is, it accepts less illegal traces of execution.

This work presents Radish, a technique to automatically generate extended finite state machines from logs of system executions. Radish combines Synoptic with Daikon, an implementation of dynamic detection of likely program invariants, to construct EFSM models. Radish model is a more accurate representation of a subject system than Synoptic model because (1) it models interplay between events and data, and (2) it uses invariant satisfiability as a heuristic to eliminate likely illegal behaviors of the system from the model.

The rest of the paper is structured as follows: Section II discusses the challenges of using Synoptic model for test case generation and motivations for Radish. Section III presents Radish technique. Section IV discusses Radish’s approach to eliminate inconsistency from the model. Section V covers related work, and Section VI concludes.

II. CHALLENGES AND MOTIVATIONS

This section uses the VerifyPin example introduced earlier to discuss the challenges of using Synoptic model for test case generation and motivate the new kind of model.

The events in VerifyPin are the following:
- **enterPin**: the user enters a pin
- **accessGranted**: the user is granted access
- **accessDenied**: the user is denied access
- **pinBlocked**: the user is blocked from getting access, regardless of whether the entered pin is correct.

VerifyPin has 2 variables: pin and pinCount. pin is an input variable, and pinCount is an internal variable that keeps track of how many further tries are allowed. pinCount is initialized to 3, is decremented by 1 each time the user enters a pin, and is reset to 3 each time the user is granted access. Synoptic model of VerifyPin is shown in figure 1.

A. Challenge 1: Derive legal abstract test cases

We want abstract test cases (i.e., paths in the model) that represent executions that can happen in the system.

Synoptic model (Figure 1) describes event sequences that cannot occur in VerifyPin, e.g., sequences with three or more consecutive enterPin-accessDenied pairs followed by accessGranted. Thus, this model may generate illegal abstract test cases. Synoptic can model simple event temporality, such as pinBlocked is never followed by accessGranted and accessDenied always precedes pinBlocked. However, event temporality in VerifyPin is also conditioned on data values the system, namely, the number of further tries allowed for the user to get access (i.e., the value of pinCount). Synoptic algorithm does not take into account the conditions on values of pinCount while constructing the model; therefore, Synoptic model does not accurately describe VerifyPin.

B. Challenge 2: Convert abstract test cases into executable test cases

Assuming that the abstract test cases are legal, we want to find the conditions of input and/or configuration values that would trigger the event sequences described in the abstract test cases.

The model in Figure 1 describes some legal event sequences, but some of those sequences can only occur under certain conditions, for example, the sequence **{enterPin, accessGranted}** occurs only when the input pin is correct. Synoptic model does not describe conditions on data values of its paths. Therefore, given an abstract test case, we do not know which input and/or configuration values to feed the system to exercise the behavior that the abstract test case describes.

C. New model: Combine temporal properties and data conditions

FSM cannot effectively model systems with interplay between data values and event sequences. The missing key properties of VerifyPin can be described by annotating edges of Synoptic model with conditions on values of pinCount and pin. The resulting model is an extended finite state machine (EFSM) — FSM with conditions on data values annotated on its edges.

One key property missing from Synoptic model of VerifyPin is accessDenied is never followed by accessGranted if and only if pinCount = 0. To discover such property, we can detect invariants of pinCount after the event accessDenied...
occurs. We will see that when accessDenied is followed by the oval-shaped enterPin, pinCount is always 0, and when accessDenied is followed by the box-shaped enterPin, pinCount is always greater than 0. Once we annotate the conditions pinCount = 0 and pinCount > 0 on the respective edges, the model no longer describes illegal event sequences.

The other two key properties missing from the original Synoptic model of VerifyPin is pin = KEY always precedes accessGranted, and pin ≠ KEY always precedes accessDenied. Similarly, to discover those properties, we can detect invariants of pin before the event accessGranted and before the event accessDenied, respectively.

The new model is shown in Figure 2.

D. Model data mutation

The new model of VerifyPin (Figure 2) is more accurate — it preserves more properties of VerifyPin than the original model. However, it still does not address challenge 1. In general, when we derive a path from a model with conditions on data values on its edges, if all conditions along the path are satisfiable, then it is likely that the path is a legal abstract test case. (Note: If the model is accurate, then a satisfiable path must be a legal abstract test case.) However, we might not know if those conditions are satisfiable, because we do not know how events in the system mutate values of the variables associated with those conditions.

We can find out whether a path is legal by choosing input and/or configuration values that satisfy the conditions along the path (assuming that the system does not modify input and configuration variables) and feeding them to the system. If the resulting execution trace matches the path, then the path is legal. Otherwise, the path maybe illegal (we cannot be certain that the path is illegal because there might be some hidden conditions on input and/or configuration values, and the values we fed the system do not satisfy those conditions).

We may want to derive abstract test cases that cover as many nodes in the model as possible. Unfortunately, unless we exhaustively search of the model and try to execute every path — by feeding the system input and/or configuration values that satisfy conditions along the path — we might not be able to find an abstract test case that includes a particular node or a sequence of nodes. This work does not propose a full solution to such problem. Instead, this work suggests an approach that could potentially enable a solution to the problem.

The approach is to model data mutation in the system, by inferring functions that describe the effects that each event has on variables. For example, the following properties describe data mutation in VerifyPin:
- enterPin always decrements pinCount by 1 when it is immediately followed by either accessGranted or accessDenied
- enterPin does not modify pinCount when it is immediately followed by pinBlocked
- accessGranted always sets pinCount to 3

Figure 3 shows the model of VerifyPin with the above properties. Suppose that we want to generate a test case that invokes the event pinBlocked, that is, we want to find a path in the model from the initial enterPin to pinBlocked that makes the condition pinCount = 0 on the incoming edge of pinBlocked true. Since we know what effect each event has on the value of pinCount, we can search for such path. To determine an efficient algorithm to find such path remains as future work.

Next section describes Radish’s technique to generate EFSM models discussed in this section.

III. GENERATING RADISH MODELS

This section describes the Radish model-inference technique. It first gives an overview of the Radish, and then discusses each step in details.
A. Overview of Radish

Radish combines 2 techniques: Synoptic and Daikon, to infer an EFSM model of a system from its execution log. Radish assumes that execution traces in an input log are sequences of events interleaving with data values. Radish parses the traces in the input log into a trace graph, then it uses Synoptic to construct an intermediate model from the trace graph. The intermediate model is an FSM model with raw data values on its edges. Radish uses Daikon to mine state invariants and data mutation functions of the events. Finally, Radish checks for model inconsistency and iteratively refines the model (and also coarsens to merge unnecessary splits) until the resulting model is consistent.

B. Constructing the trace graph

Radish constructs a model from a set of system execution traces. It takes as input a log file containing the traces and a set of regular expressions to parse the log. Radish assumes that the traces are sequences of event instances interleaving with data values. An event instance is represented by a name or event type, e.g., enterPin. Data values are represented by a set of variable-value pairs, e.g., \{pin = 123, pinCount = 2\}. The traces in the input log are parsed into a trace graph. A trace graph is a set of linear graphs with nodes representing event instances, and edges representing the ordering of those event instances and capturing data values. Figure 4 shows a trace graph with two traces.

C. Intermediate model

Radish uses Synoptic algorithm to construct an intermediate model from the trace graph. This intermediate model is simply a partition graph of the trace graph. Given a partitioning of the original nodes, each node in the model is one partition. Directed edges in the model are formed through existential abstraction. That is, a directed edge between two nodes indicates that there exists a pair of event instances in the corresponding partitions that are connected by an edge in the trace graph. Therefore, each edge in the model is associated with a set of sets of variable-value pairs. Each set of variable-value pairs comes from an edge between 2 event instances in the trace graph.

D. Mining state invariants from the intermediate model

In the intermediate model, each edge represents a state in the system. Radish uses Daikon to mine a set of invariants associating variables of each state. An invariant can be of a single variable (e.g., \(x > 0\)) or relate multiple variables (e.g., \(x < y\)).

Figure 5 shows a model of VerifyPin with state invariants.

E. Mining mutation functions from the trace graph

The goal of this step is to infer functions that describe the effects that each event has on variables. For each distinct sequence \(s = \{a, b, c\}\) of three events in the trace graph, Radish finds all sequences of three events from the trace graph that are identical to \(s\), then it uses Daikon to mine invariants that relate variables on the edge \((b, c)\) to variables on the edge \((a, b)\) across all of these sequences.

The resulting model in Figure 6 captures interplay between event sequences and data values. It models the effect that data values have on event temporality, i.e., path constraints. And it models the effect the events have on data values, i.e., data mutation.
Fig. 6. A model of VerifyPin with state invariants. This model captures the effect data values have on event sequences, i.e., path constraints.

Fig. 7. Radish model of VerifyPin. This model captures interplay between event sequences and data values.

F. Refining model to eliminate contradictions

After the above process of constructing Radish model, the resulting model may have contradictions. Since Synoptic is a generative model-inference algorithm, a model it produces may contain unobserved behaviors, i.e., paths that do not match any trace in the trace graph — synthetic paths. Synoptic algorithm does not reason about data conditions. Therefore, once Radish augments Synoptic model with data conditions, there is no guarantee that data conditions along a synthetic path are satisfiable.

This work propose to use invariant satisfiability as a heuristic to eliminate likely illegal behaviors from Radish model. Next section describes this heuristic in details and discusses refine-

Fig. 8. A contradictory path. Suppose that no events modify variable $x$. The path $\{a, b, c\}$ is synthetic — it was never observed in the log. The invariant of the variable $x$ between the events $a$ and $b$ mined from the traces is $x > 0$, and the invariant of $x$ between $b$ and $c$ is $x = 0$.

IV. ELIMINATE INCONSISTENCY

The model that we derive using Radish may have paths that violate certain data consistency constraints. In our approach, we define consistency in terms of invariant satisfiability. For example, a path $P$ in the model may contains a subpath $P' = \{p_0, \ldots, p_n\}$ such that the set of invariants $i_1$ on the edge from $p_0$ to $p_1$ and the set of invariants $i_n$ on the edge from $p_{n-1}$ to $p_n$ cannot be both satisfied at the same time. That is, there is no legal assignment of variables to values that can make $i_1$ and $i_n$ both hold true. Note, however, that this assumes that the events along the path $P'$, that is, $p_0, \ldots, p_n$, do not make modifications to the variables associated with invariants $i_1$ and $i_n$. This raises an issue: how do we remove the inconsistency (i.e., eliminate the subpath $P'$ from the model).

A. Path consistency

A path in the model is consistent if and only if the conjunction of all invariants of unmodified variables along the path is satisfiable. Note that only synthetic paths can be inconsistent.

B. Refinement

We use path consistency as a heuristic for determining whether a synthetic path represents a legal behavior of the subject system. We want to eliminate illegal behaviors from the model, thus, we eliminate all of its inconsistent paths. Starting with a model that Synoptic infers, Radish refines (i.e., splits) partitions until it reaches a model whose all paths are consistent. Radish outputs the model when it is unable to refine it any further.

The refinement goal of Radish is to pick a minimal sequence of splits, so that the resulting graph is the coarsest graph that does not have any inconsistency. This problem is NP-hard, so an efficient algorithm might not yield the optimal result.
Radish performs splits as long as there exists some path that is inconsistent. It uses an SMT solver to check whether the conjunction of all invariants of unmodified variables along a path is satisfiable. Radish depth-first traverses the model starting from the initial partition, and uses SMT solver to check, at every partition it visits, if the current path — the path consisting entirely of visited partitions, from the initial partition to the currently visited partition — is consistent. If an inconsistent path \( P \) is found, we want to eliminate the inconsistency in \( P \) from the model. Since we want to minimize the number of splits, Radish finds the shortest inconsistent subpath \( P' \) of \( P \) to eliminate. The shortest inconsistent subpath of \( P \) is the shortest inconsistent suffix of \( P \).

Once identified a path \( P' \) to eliminate, Radish refines one or more partitions to eliminate \( P' \). Let \( P' = \{a, b, c, \ldots, m, n\} \), \( i_b \) = conjunction of all invariants of unmodified variables on \( (a, b) \) edge, and \( i_e \) = conjunction of all invariants of unmodified variables on \( (m, n) \) edge. We know that \( i_b \land i_e \) is unsatisfiable. Radish starts by splitting \( b \) into 2 partitions: \( b_1 \) and \( b_2 \).

If the sequence \( \{e_a, e_b, e_c\} \) is in the trace graph, then \( P \) still remains after splitting \( b \). In this case, Radish splits \( c \) next. Similarly, if the sequence \( \{e_a, e_b, e_c, e_d\} \) is in the trace graph, then \( P \) still remains after splitting \( c \). In this case, Radish splits \( d \) next, and so on. Until partition \( i \) is splitted and the sequence \( \{e_a, e_b, e_c, \ldots, e_i\} \) is not in the trace graph. Note that such partition \( i \) always exists, and it may be \( m \) because the sequence \( \{e_a, e_b, e_c, \ldots, e_m, e_n\} \) must not be in the trace graph.

**C. Coarsening**

At this point, \( P' \) is eliminated, but Radish may end up refining more than it needs to. When this happens, the model will contain partitions that can be merged without violating path consistency. After eliminating \( P' \), Radish coarsens the model to merge such partitions.

After refinement and coarsening of each path, Radish must re-calculate invariants on all edges that were splitted. Next, Radish repeats model check for path inconsistency and performs refinement and coarsening, until there it reaches a model that is consistent.

**D. Event purity**

The definition of path consistency only involves invariants of unmodified variables. Radish approximates which variables are unmodified by a particular event based on the observed execution traces. Radish assumes that an event \( e \) is pure with respect to, i.e., does not modify values of, variables \( x_1, \ldots, x_n \) if for every instance of \( e \) in the trace graph, the values of \( x_1, \ldots, x_n \) on the outgoing edge of \( e \) is the same as the values of \( x_1, \ldots, x_n \) on the incoming edge of \( e \).

**V. RELATED WORK**

The techniques that this work uses directly are Synoptic and Daikon. Synoptic [1] is a tool that infers a concise and accurate system model in a form of FSM model from log. Daikon [2] is an implementation of dynamic detection of likely program invariants. GK-Tails [3] infers EFSM models from interaction traces.

**VI. CONCLUSION**

Radish is a technique to automatically generate EFSM models of systems from their execution logs. Radish uses Synoptic, an FSM model inference technique, and Daikon, a likely program invariant detection technique, to reason about dependencies between event temporality and data values in a subject system. This work also proposes a strategy to control generalizations in model inference technique such as Synoptic. It uses invariant satisfiability as a heuristic for determining illegal predicted behaviors in the model, and for guiding model refinement to eliminate likely false properties. Radish models have the potential to be used for automated test case generation. Determining whether Radish models are useful for test case generation remains as future work.

**REFERENCES**

