Temporal Specification Mining for Anomaly Analysis

Farn Wang, Jung-Hsuan Wu, Chung-Hao Huang, Cheng-Chieh Chang, and Chung-Cheng Li

Dept. of Electrical Engineering & Graduate Institute of Electronic Engineering
National Taiwan University
Nr.1, Sec. 4, Roosevelt Rd. Taipei, Taiwan 106, ROC
`http://cc.ee.ntu.edu.tw/~farn/`

Abstract. [1] We investigate how to use specification mining techniques for program anomaly analysis. We assume the input of positive traces (with- out execution anomalies) and negative traces (with execution anomalies). We then partition the traces into the following clusters: a positive cluster that contains all positive traces and some negative clusters according to the characteristics of trace anomalies. We present techniques for learn- ing temporal properties in Linear Temporal Logic with finite trace se- mantics (FLTL). We propose to mine FLTL properties that distinguish the negative clusters from the positive cluster. We experiment with 5 Android applications from Google Code and Google Play with traces of GUI events and crashes as the target anomaly. The report of FLTL properties with high support or confidence reveal the temporal patterns in GUI traces that cause the crashes. The performance data also shows that the clustering of negative traces indeed enhances the accuracy in mining meaningful temporal properties for test verdict prediction.

Keywords: Specification mining, FLTL, Clustering, Program trace, Android.

1 Introduction

Nowadays, many software programs are developed with rapid life-cycles. For example, various applications running on mobile devices emerge with billion times of monthly downloads. Most of these applications keep frequent bug fixing, performance improvement, and feature enhancement during their life cycles. As a result, to keep a correct and up-to-date specification of such an application of a non-trivial size has become a daunting challenge. One promising technology to address this problem is *specification mining* [2,11,20] that extracts specifications from program execution *traces* collected via program code instrumentation. In this work, we investigate how to mine temporal specifications for the diagnosis of anomalies in traces.

[1] The work is partially supported by NSC, IIS and Academia Sinica.

© Springer International Publishing Switzerland 2013

Our work assumes a test oracle that issues verdicts to execution traces. A trace is *positive* if it is labeled with the verdict of no anomaly; otherwise, it is *negative*. A specification property is *positive* if it is expected of a correct implementation; otherwise, it is *negative*. Previous work in specification mining focuses on mining positive properties from all traces. To help in diagnosis, we need consider the following issues.

- There are usually more than one anomaly with different root causes exhibited in the traces. Thus the behavior patterns of an anomaly may not be shared by all the negative traces and could be mined with low support. To counter this problem, we propose to partition the negative traces to clus- ters according to trace characteristics related to anomalies. Our experiment shows that mining the individual clusters of the negative traces indeed helped us in mining useful and meaningful negative specification properties.
- To help in diagnosis, we need to consider what patterns can be typical of anomalies in traces. Since most bugs are reported as program traces, we adopt *linear temporal logic* with finite trace semantics (FLTL) as our specification language for behavior patterns of traces. Moreover, we identified five types of FLTL formulas that are typical of trace anomalies. In our experiment, the five types helped in mining FLTL properties for recognizing trace anomalies.
- Finally, we want to avoid mining effective FLTL properties shared by both negative and positive traces. This is easily done by subtracting the mined properties of the positive traces from those of the negative ones.

We have implemented our ideas and experimented with 5 Android applications from Google Code and Google Play. The preliminary experiment data shows the promise of our techniques.

The organization of this paper is structured as follows. Section 2 compares this work with related work in the literature. Section 3 reviews the background knowledge. Section 4 respectively present our diagnosis property mining framework and the five types of FLTL properties that we target to mine. Section 6 explains our mining algorithms for the five types of FLTL properties. In Section 7, we explain our implementation on the Android platform. Section 8 reports our experiment with five benchmarks. Section 9 is the conclusion.

2 Related Work

Several approaches about specification mining have been proposed in the literatures. Ernst et al. developed a tool, Daikon, for automatic deduction of likely invariants involving program variables from dynamic traces [4]. Specifically, Daikon mines arithmetic relationships, such as $x \geq y$, that hold at specific statements. Our work is for mining FLTL properties that relate several program states.

Ammons et al. introduced an automata learning framework to learn *nondeterministic finite-state automata* (*NFSA*) from program traces [2]. The programs traces are collected from an instrumented program that is well-debugged to reveal strong hints of correct behaviors. In our work, we do not assume such a golden program.

Dallmeier et al. proposed an incremental approach to mine normal program behaviors [3]. Starting with a set of program traces, they construct an initial automata and generate more test cases based on this automata to explore more execution space. These test cases either end in a legal state or raise an exception. The test case execution leads to repetitive enrichment to the automata. This procedure repeats until no further test cases can be generated.

Lorenzoli et al. presented a technique to automatically generate extended finite-state machines from program traces [14]. By labelling transitions with conditions on data values, an extended finite-state machine can model the interplay between data values and component interactions.

In contrast to the work on mining automata, we focus on mining FLTL properties which are more appropriate in expressing behavior patterns relating events far apart and in expressing liveness properties than automata.

Yang et al. developed a tool, Perracotta, to discover temporal properties between only two events in traces about application program interface [20].

Lo et al. proposed several approaches to mine program specifications from execution traces [13,12]. In [13], they mined universal Live Sequence Chart which captures the inter-object behaviors of multiple events in arbitrary length. In [9], they mined recurrent rules in the form of "whenever a series of precedent events occurs, eventually a series of consequent event occurs". These are similar to one type of FLTL properties that we target to mine. In our experiment, we found that other types of properties are also important in anomaly diagnosis. In [12], they mined past-time temporal rules while our TLTL templates express future time temporal properties.

Since the target SUT may suffer from different anomalies, mining directly from the collected program negative traces may be ineffective due to the interference of different anomalies. Lo et al. clustered program traces by their similarity between pairs of data item [10]. Our works cluster negative traces by the program subroutines of topmost exception call stack frame.

3 Preliminaries

3.1 Model of Finite Program Execution Traces

A *trace* is intuitively a finite linear sequence of events. For convenience, we use $|\theta|$ to denote the length of a trace θ. Rigorously, a trace θ is a function from $[0, |\theta| - 1]$ to a set of events. For any integer i and j with $0 \leq i \leq \jmath \leq |\theta|$, we let $\theta[i...j] = \theta(i)\theta(i+1)\theta(i+2)...\theta(j)$. We also use θ_i to denote the i^{th} suffix of θ, i.e., $\theta_i = \theta(i)\theta(i+1)\theta(i+2)...\theta(|\theta| - 1)$.

3.2 Linear Temporal Logic with Semantics on Finite Trace

In 1977, Pnueli proposed linear temporal logic (LTL) as a formal language for expressing and reasoning about the behavioral properties of parallel programs and reactive systems[15]. In this work, we focus on the analysis of finite-length program traces. Therefore we adopt FLTL [6,18,7]. FLTL formulae (ranged over by $\varphi, \psi, ...$) on a finite set P of events are of the following syntax.

$$\varphi ::= true \mid e \mid \neg\varphi \mid \varphi \wedge \psi \mid X\varphi \mid \varphi U \psi$$

Here $e \in P$ is an event in a program trace. Operators $\neg$ and $\wedge$ are Boolean negation and conjunction respectively. X and U are temporal operators next and until respectively.

Boolean connectives such as $\vee, \rightarrow$ and $\leftrightarrow$ are derived operators from $\neg$ and $\wedge$. Other useful shorthands include $false \equiv \neg true$, $F\varphi \equiv trueU\varphi$ (the *eventually* operator), and $G\varphi \equiv \neg(trueU\neg\varphi)$ (the *always* operator).

The satisfaction of an FLTL formula φ by a finite program trace θ, written $\theta \models \varphi$, is defined inductively as follows.

$$\begin{array}{lll}
\theta \models true & \text{iff} & true \\
\theta \models e & \text{iff} & |\theta| > 0 \wedge \theta(0) = e \\
\theta \models \neg\varphi & \text{iff} & \theta \not\models \varphi \\
\theta \models \varphi \wedge \psi & \text{iff} & \theta \models \varphi \wedge \theta \models \psi \\
\theta \models X\varphi & \text{iff} & |\theta| > 0 \wedge \theta_1 \models \varphi \\
\theta \models \varphi U \psi & \text{iff} & \exists k \leq |\theta|, (\theta_k \models \psi \wedge \forall j \in [0, k-1], \theta_j \models \varphi)
\end{array}$$

Intuitively, $X\varphi$ says that φ is true in the next position. A formula $\varphi U \psi$ means that ψ is true either now or in the future and φ holds since now until that moment.

Example 1. A program trace $\theta = s_0 e_0 s_1 e_1 s_0 e_2 s_2 e_3 s_0$ would satisfy property $\varphi = G((s_0 \wedge Xe_2) \rightarrow XXs_2)$. This property can be interpreted as that state s_0 with input e_2 will always transit to state s_2. □

3.3 Association Rule Mining

In the field of data mining, association rule mining is one of the most important and well researched techniques[8,21]. It was first introduced by Agrawal [1] and aims to extract interesting correlations, associations, frequent patterns among sets of items in a dataset. Since then, it has been widely used in various areas, such as telecommunication networks, market and risk management, inventory control etc. Given a set I of items of a record, an *association rule* is of the form $X \mapsto Y$, where $X, Y \subset I$ and $X \cap Y = \emptyset$, that specifies that the occurrences of all items in X implies that the occurrences of all items in Y is also likely to occur.

Example 2. In daily supermarket transactions, a transaction could consist of buying tomatoes, cheeses, noodles, eggs, etc. The rule $\{$tomatoes, noodles$\} \mapsto \{$eggs$\}$ mined from transactions would indicate that if a customer buys tomatoes and noodles together, he is likely to also buy eggs. □

There are two important basic measures to evaluate how likely an association rule is. They are *support* and *confidence*. Support is the statistical significance of an association rule. It is defined as the percentage/fraction of records that contain all items in $X \cup Y$ to the total number of records in the dataset, i.e.,

$$support(X \mapsto Y) = \frac{\text{Number of records in dataset with all items in } X \cup Y}{\text{Total number of records in dataset}}$$

Intuitively, a high support value is an evidence that the related rule is significant. Users can specify a support value as a threshold for mining significant rules. Only association rules with supports higher than the threshold are then reported.

However, sometimes an association rule with low supports could also be interesting. For example, in the supermarket case mentioned above, transactions with high price items could be rare. But association rules related to these expensive items are also important to the retailer. Therefore, another measure, confidence, has also been used to deal with this situation. Confidence is a measure of strength of the association rules. It is defined as the percentage/fraction of the number of records that contain $X \cup Y$ to the total number of records that contain X. Formally, confidence is calculated by the following formula:

$$confidence(X \mapsto Y) = \frac{\text{Number of records in dataset with all items in } X \cup Y}{\text{Number of records in dataset with all items in X}}$$

If the confidence of the association rule $X \mapsto Y$ is 80%, it means that 80% of the records that contain X also contain Y. Similarly, users can also prescribe a confidence threshold to ensure that only interesting rules are reported.

4 Anomaly Specification Mining Framework

In order to mine FLTL properties from the negative traces, we propose the framework showed in Figure 1. At the top-left corner, we accept program execution traces. After normalization, we may change the traces to a format suitable for efficient and accurate processing. Specifically, we only keep the last state event in a sequence of consecutive state events so that all normalized traces are strict alternations of state and input events.

We assume that there is a test oracle (module verdict) that issues verdicts to the normalized traces. The test oracle can be a human engineer or can be a program that checks the traces against a formal specification. Traces without anomalies are labeled 'pass' and treated as positive traces. The other traces are labeled 'fail' and treated as negative traces. For example, in our experiment, we target system exceptions as anomalies. If there is a system exception during the trace execution, the trace will be labelled as a fail trace.

Since we use data mining technic to extract FLTL properties from fail traces, we want to make sure that the bug-trigger behaviors come into the miner's notice. For this reason, we try to class the fail traces into different clusters according to the bugs they triggered, automatically. After that, the cause of a bug should appear overwhelmingly in the cluster it belongs. In this work, the fail (negative) traces are partitioned (by module cluster) into clusters, say C_1 through C_k, according to trace characteristics related to anomalies. For example,

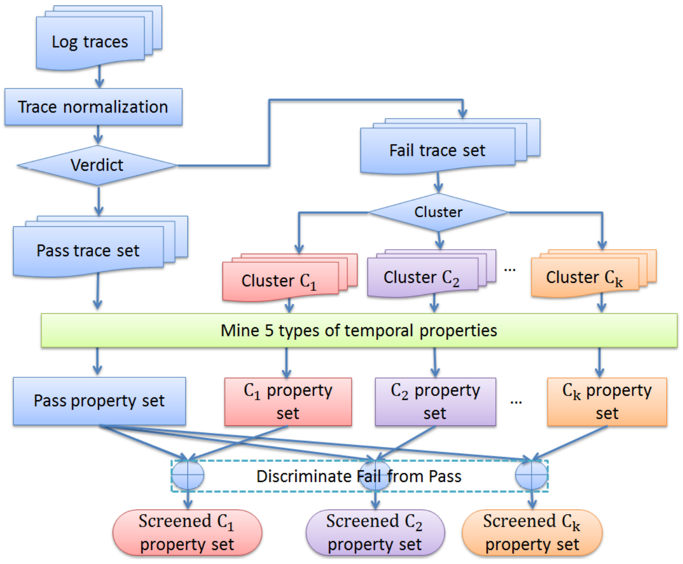

Fig. 1. The anomaly specification mining framework

in our experiment, since we view system exceptions as evidences of trace anomalies, we use the contents of the exception call stack to partition the negative traces. That is negative traces are partitioned into the same cluster, if the top-most stack frames issued by the SUT are with the same subroutine. Ideally, each cluster should contain negative traces with anomalies of the same root cause.

For convenience, we assume that the set of positive traces is in cluster C_0. Then we apply our FLTL property mining algorithm to cluster C_0 through C_k and get FLTL property sets R_0 through R_k respectively. If the mining algorithm works well, then for each $i \in [1, k]$, R_i should contain FLTL properties that sufficiently recognize the anomalies of the root cause for C_i.

Finally, to avoid producing diagnosis properties also satisfied by the positive traces, we only output $R_1 \backslash R_0$ (screened C_1 property set) through $R_k \backslash R_0$ (screened C_k property set), i.e., the set subtraction of R_1 through R_k by R_0.

5 Target FLTL Templates

For this work, we applied our techniques to the *graphical user-interface* (*GUI*) event traces of Android applications. There are 2 types of events in the execution traces we collected from Android applications. The first type, called *state* events, consists of durational events representing values of screen attributes. The second type, called *input* events, consists of instantaneous user input via the GUI components on the screen. For example, the display of a button on the screen is a state event. A finger stroke on the screen is an input event. We use E to denote the set of input events while S to denote the set of state events.

Traditional mining techniques are usually bottom-up and deduce complex properties from simple formulas. Such an approach is not guided and is usually drowned in the sea of small properties without any hope for mining larger and complex properties. For example, we may mine property ϕ_A as a positive evidence and ϕ_B as a negative evidence of a cluster respectively. Then together, this cluster may satisfy $\neg(\phi_A \rightarrow \phi_B)$ with high confidence. If ϕ_A and ϕ_B are already complex FLTL properties, then it is not likely that we can mine $\neg(\phi_A \rightarrow \phi_B)$ in a bottom-up style without proper guidance. In this work, we propose to guide the mining procedure via property templates whose logical combinations can capture some complex and maybe interesting properties. Considering most bugs in Android applications can be triggered by a series of screen touch behaviors or by a screen touch under certain system conditions, we defined five target FLTL templates that, we believe, have good opportunity to find the root cause of bugs.

In the following, we explain the detail of the five types of FLTL properties. Please be reminded that these five types have been used effectively in our experiment for our purpose. It is certainly possible that in the future, more types are designed for other experiments.

5.1 Γ_1: Nested Eventuality Properties

In previous work, temporal properties with very restricted syntax are mined to contain the search space for mined properties [4,11,20]. Such approaches usually can deduce piecewise behavior patterns of the positive traces. In contrast, a negative trace in a bug report may contain a long sequence of events leading to the exhibition of an anomaly. Specifically, for GUI event traces, a trace anomaly usually is triggered by an interleaved sequence of state events and input events. Thus, intuitively, a sequence of interleaved events naturally matches nested FLTL eventually properties. For example, the following FLTL property specifies the anomaly of the tapping of the "next" button at screen $Page_1$ followed by the tapping of "yes" button at screen $Page_2$ that leads to screen $Page_3$.

$$F(Page_1 \wedge X(next \wedge XF(Page_2 \wedge X(yes \wedge XF(Page_3)))))$$

In general, a trace anomaly of this type can extend to any depth of nesting. Specifically, we can use the following syntax to define this type.

$$\varphi ::= F(s) \mid F(s \wedge X(e \wedge X\varphi))$$

Here $s \in S$ and $e \in E$. We let Γ_1 be the set of FLTL properties of this type.

5.2 Γ_2: Nested Conditional Next Properties

Sometimes, a strict sequence of events must happen for the setup of certain procedures. In such a procedure, if one step is wrong, an anomaly may occur. For example, we may want to say that after the query for password, if the user taps either button 'cancel' or button 'prev' (for previous page), the login should not be successful. This can be expressed with the following two properties.

$$G(\mathit{login\_display} \rightarrow X(\mathit{cancel} \rightarrow X(\mathit{login\_succ})))$$
$$G(\mathit{login\_display} \rightarrow X(\mathit{prev} \rightarrow X(\mathit{login\_succ})))$$

Here *login_display* is a state event for the screen of the login page that queries an account name and a password. State event *login_succ* flags the success of the login procedure. As can be seen, several properties of this type together may be used to express the 'or' concept in detecting events. For example, the following two traces both satisfy the two properties.

login_display cancel login_succ logout login_display prev login_succ
login_display prev login_succ logout login_display cancel login_succ

Here we use *logout* to denote the input event for logging out.

In general, a trace anomaly of this type can extend to any depth of nesting. Specifically, a property φ of this type is of the following syntax.

$$\varphi ::= G(s \rightarrow X(e \rightarrow X(\psi)))$$
$$\psi ::= s \mid s \rightarrow X(e \rightarrow X(\psi))$$

Here $s \in S$ and $e \in E$. We let Γ_2 be the set of FLTL properties of this type.

5.3 Γ_3: Nested Conditional Eventuality Properties

In Γ_2, the strict sequence is in lock-steps and one event in the sequence must be followed by another in the sequence. Sometimes, especially for multi-thread or distributed programs, there could be some irrelevant events from other threads or applications in the traces. Thus, by replacing the 'X' operators before the state events with the XF operators (eventuality), we can rewrite properties in Γ_2 for more flexibility in the sequence. In general, a trace anomaly of this type can extend to any depth of nesting. Specifically, a property φ of this type is of the following syntax.

$$\varphi ::= G(s \rightarrow X(e \rightarrow XF(\psi)))$$
$$\psi ::= s \mid s \rightarrow X(e \rightarrow XF(\psi))$$

Here $s \in S$ and $e \in E$. We let Γ_3 be the set of FLTL properties of this type.

5.4 Γ_4: For Uninitialization Anomaly

We also considered the anomalies with improper initializations. For example, program may access an object before it is instantiated. Such an anomaly can be specified as: $(\neg \mathit{new})U\mathit{read}$. The syntax of such properties is the following.

$$\varphi ::= (\neg s_0)U(s_1 \wedge Xe_1) \mid (\neg e_0)U(s_1 \wedge Xe_1)$$

Here $s_0, s_1 \in S$ and $e_0, e_1 \in E$. We use Γ_4 to denote the set of all such properties. There is no expansion to such properties in this work.

Algorithm 1. Expand(φ, d)

```
1: if $d$ is greater than the prescribed expansion depth then
2:     Return.
3: end if
4: for each $e \in E$ and $s \in S$ do
5:     if $\frac{|\{\theta|\theta \in C, \theta \models \varphi+e+s\}|}{|C|} \geq t$ then
6:         Report $\varphi + e + s$ and call Expand($\varphi + e + s$, $d + 1$)
7:     end if
8: end for
```

5.5 Γ_5: For Nested Starvation Anomaly

Some anomalies exhibit the denial of services after some event sequences have been observed. For example, an anomaly that two consecutive reads to a page make the page no longer readable can be specified with the following property.

$$G(page\_loaded \rightarrow X(read \rightarrow X(page\_loaded \rightarrow X(read \rightarrow X(G\neg page\_loaded)))))$$

The syntax of such a property φ is of the following.

$$\begin{aligned} \varphi &::= G(s \rightarrow X(e \rightarrow X(\psi))) \\ \psi &::= G\neg s \mid G\neg e \mid s \rightarrow X(e \rightarrow X(\psi)) \end{aligned}$$

Here $s \in S$ and $e \in E$. We let Γ_5 be the set of FLTL properties of this type.

6 Mining Algorithms

In the following, we first define how to expand properties of a type and then use the expansion operator to explore the space of FLTL properties up to a limit of nesting of the templates prescribed by the users. We assume that the users have prescribed a threshold t of both support and confidence. Only properties with support (or confidence) no less than t will be reported and used for further expansion. In the following, we assume that we are given a trace cluster C.

6.1 Mining Algorithm for Γ_1 Properties

Given a $\varphi \in \Gamma_1$ of the form $\varphi = F(s_1 \wedge X(e_1 \wedge XF(\ldots XF(s_n)\ldots)))$, we let $\varphi + e + s$ denote the expansion of φ with one more nesting of the eventuality of an input e followed by a state s. Specifically, $\varphi + e + s$ represents

$$F(s_1 \wedge X(e_1 \wedge XF(\ldots XF(s_n \wedge X(e \wedge XF(s)))\ldots))).$$

Given a state event s and an input event e, we only use the support of $\varphi+e+s$, defined as $\frac{|\{\theta|\theta \in C, \theta \models \varphi+e+s\}|}{|C|}$, to evaluate the property. Then we use algorithm 1 to recursively expand the properties of a type. The mining algorithm starts by calling Expand$(F(s \wedge X(e \wedge XF(s'))), 0)$ for all $s, s' \in S$ and $e \in E$.

6.2 Mining Algorithm for Γ_2 Properties

Given a $\varphi \in \Gamma_2$ of the form: $\varphi = G(s_1 \rightarrow X(e_1 \rightarrow X(\ldots X(s_n)\ldots)))$, we let $\varphi + e + s$ denote the expansion of φ with one more nesting of the eventuality of an input e followed by a state s. Specifically, $\varphi + e + s$ represents

$$G(s_1 \rightarrow X(e_1 \rightarrow X(\ldots X(s_n \rightarrow X(e \rightarrow X(s)))\ldots))).$$

According to the literature, there can be many different granularities in defining the confidence of such properties. Here we want to use a granularity smaller than traces. That is, we want to count how many times a property in Γ_2 is honored in a trace. For this purpose, we need to define the following concepts. Given a property $\varphi \in \Gamma_2$, we let $\boxplus(\varphi)$ be the sequence of events in φ listed in the order that they appear in φ. For example, $\boxplus(G(a \rightarrow X(b \rightarrow X(c)))) = abc$.

Given a property $\varphi \in \Gamma_2$, a state event s, and an input event e, we only use the confidence of properties of this type to evaluate them. The confidence of $\varphi + e + s$ is defined as $\frac{|\{(\theta,i,j)|\theta \in C, 0 \leq i \leq j < |\theta|, \theta[i\ldots j] = \boxplus(\varphi)es\}|}{|\{(\theta,i,j)|\theta \in C, 0 \leq i \leq j < |\theta|, \theta[i\ldots j] = \boxplus(\varphi)e\}|}$. The mining algorithm is basically Algorithm 1 except that line (5) is replaced with

$$\textbf{if } \frac{|\{(\theta,i)|\theta \in C, 0 \leq i \leq j < |\theta|, \theta[i\ldots j] = \boxplus(\varphi)es\}|}{|\{(\theta,i)|\theta \in C, 0 \leq i \leq j < |\theta|, \theta[i\ldots j] = \boxplus(\varphi))e\}|} \geq t$$

The mining algorithm starts by calling Expand$(G(s \rightarrow X(e \rightarrow X(s'))), 0)$ for all $s, s' \in S$ and $e \in E$.

6.3 Mining Algorithm for Γ_3 Properties

Given a $\varphi \in \Gamma_3$ of the form: $\varphi = G(s_1 \rightarrow X(e_1 \rightarrow XF(\ldots XF(s_n)\ldots)))$, we also let $\varphi + e + s$ denote the expansion of φ with one more nesting of the eventuality of an input e followed by a state s. Specifically, $\varphi + e + s$ represents

$$G(s_1 \rightarrow X(e_1 \rightarrow XF(\ldots XF(s_n \rightarrow X(e \rightarrow XF(s)))\ldots))).$$

We also let $\boxtimes(\varphi)$ be the regular language:

$$s_1 e_2 (S \cup E)^* s_2 e_2 (S \cup E)^* \ldots (S \cup E)^* s_{n-1} e_{n-1} (S \cup E)^* s_n.$$

Here $(S \cup E)^*$ represents the set of sequences with only zero or more events in $S \cup E$. That is, $\boxtimes(\varphi)$ is obtained from $\boxplus(\varphi)$ by inserting $(S \cup E)^*$ before every state event except the first one. For example,

$$\boxtimes(G(a \rightarrow X(b \rightarrow XF(c)))) = ab(S \cup E)^* c$$

We use $\langle L \rangle$ to denote the set of strings in a regular language L.

Given a property $\varphi \in \Gamma_3$, a state event s, and an input event e, we only use the confidence of properties of this type to evaluate them. The confidence of $\varphi + e + s$ is defined as $\frac{|\{(\theta,i,j)|\theta \in C, 0 \leq i \leq j < |\theta|, \theta[i\ldots j] \in \langle \boxtimes(\varphi)e(S \cup E)^* s\rangle\}|}{|\{(\theta,i,j)|\theta \in C, 0 \leq i \leq j < |\theta|, \theta[i\ldots j] \in \langle \boxtimes(\varphi)e\rangle\}|}$. The mining algorithm is basically Algorithm 1 except that line (5) is replaced with

$$\textbf{if } \frac{|\{(\theta,i,j)|\theta \in C, 0 \leq i \leq j < |\theta|, \theta[i\ldots j] \in \langle \boxtimes(\varphi)e(S \cup E)^* s\rangle\}|}{|\{(\theta,i,j)|\theta \in C, 0 \leq i \leq j < |\theta|, \theta[i\ldots j] \in \langle \boxtimes(\varphi)e\rangle\}|} \geq t$$

The mining algorithm starts by calling Expand$(G(s \rightarrow X(e \rightarrow XF(s'))), 0)$ for all $s, s' \in S$ and $e \in E$.

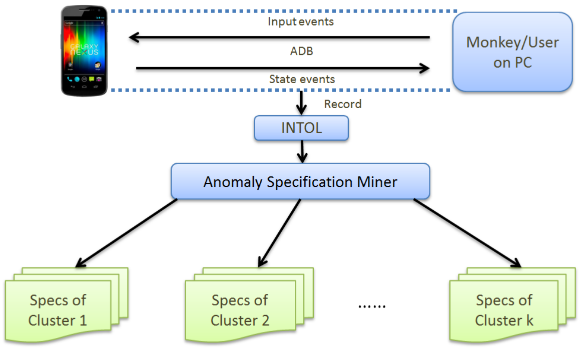

Fig. 2. The implementation

6.4 Mining Algorithm for Γ_4 Properties

This mining algorithm is pretty much Algorithm 1. We also use the support to evaluate such properties as in line (5) of algorithm 1. Due to the straightforwardness, we omit the explanation.

6.5 Mining Algorithm for Γ_5 Properties

Given a $\varphi \in \Gamma_5$ of the form: $\varphi = G(s_1 \rightarrow X(e_1 \rightarrow X(\ldots X(s_n \rightarrow X(e_n \rightarrow X(G\neg s)))\ldots)))$, we let $\varphi + s_{n+1} + e_{n+1}$ denote the expansion of φ as

$$G(s_1 \rightarrow X(e_1 \rightarrow X(\ldots X(s_n \rightarrow X(e_n \rightarrow X(s_{n+1} \rightarrow X(e_{n+1} \rightarrow X(G\neg s)))))\ldots))).$$

We also let $\boxdot(\varphi)$ be the sequence of events $s_0e_0\ldots s_ne_n$ and $\boxminus(\varphi)$ be the regular language $s_0e_0\ldots s_ne_n(S\cup E\backslash s)^*$. Given a property $\varphi \in \Gamma_5$, a state s and an event e, we define the confidence of $\varphi+e+s$ as $\frac{|\{(\theta,i)|\theta\in C,0\leq i<|\theta|,\theta[i\ldots|\theta|)\in\langle\boxminus(\varphi+s+e)\rangle\}|}{|\{(\theta,i,j)|\theta\in C,0\leq i\leq j<|\theta|,\theta[i\ldots j]=\boxdot(\varphi+s+e)\}|}$. The mining algorithm is basically Algorithm 1 except that line (5) is replaced with

$$\textbf{if } \frac{|\{(\theta,i)|\theta\in C,0\leq i<|\theta|,\theta[i\ldots|\theta|)\in\langle\boxminus(\varphi+s+e)\rangle\}|}{|\{(\theta,i,j)|\theta\in C,0\leq i\leq j<|\theta|,\theta[i\ldots j]=\boxdot(\varphi+s+e)\}|} \geq t$$

The mining algorithm starts by calling Expand($G(\neg s)$,0) for all $s \in S$.

7 Implementation

We have implemented our mining tool for Android applications. As showed in Figure 2, we first instrumented Android source code of Ice Cream Sandwich 4.0.3 version with Intelligent Test Oracle Library (InTOL) [19] to intercept GUI state events and input events, such as *onTouch*, *onKeyDown*, *onKeyUp*, etc. Then we use Monkey, a pseudo-random stream of user events generator, from Android SDK to automatically exercise an android application under test to collect the execution traces. Then through the framework proposed in Section 4, the temporal properties for each cluster are mined from the normalized traces.

The components in Figure 2 are explained as follows:

- Android Debug Bridge (ADB) comes with Google Android SDK. It is a command-line tool that communicates with an Android virtual device or connected Android mobile device. It can manage the state of the Android system, run shell commands, copy files to or from a device, and etc.
- Monkey also comes with Android SDK. It generates pseudo-random events, such as click-buttons, touches, gestures, and system-level events. Through ADB, Monkey can feed input sequences of specific lengths to the applications under test.
- Intelligent Test Oracle Library (InTOL) [19] is a tool library for the convenient and flexible collection of program traces. It can record the input events and system GUI states in traces. In addition, when the system crashes, InTOL can also log the crash event and mark the trace with 'fail.'

We also implemented a procedure that normalizes identifiers of the events and UI components in the traces. For trace-specific identifiers, we normalize them according to their order of occurrence. For process and platform-specific identifiers, we change them to constants according to our knowledge of Android.

8 Experiment

The experiment is deployed on a Samsung Galaxy Nexus i9025 running Android Ice-cream Sandwich 4.0.3 and a PC running ubuntu 10.04. Program traces are collected on the Galaxy Nexus and analyzed on the PC. In collecting the traces, we use Monkey to exercise each benchmark 3000 times with 300 input stimulus per exercising. To simulate a normal user's pace in operating an Android application, Monkey injects 10 events per second. If an application crashes in an exercising, the trace will be ended immediately with a fail verdict. Each benchmark takes approximately 20 hours on average to collect 3000 traces.

We use expansion depth of 6 in all mining processes. In the following, we first introduce our benchmarks. Then we report the performance of our techniques when used for test verdict prediction. Finally we examine an example property mined via our tool and argue for their values.

8.1 Benchmarks

We have five Android applications from Google Code and Google Play as our benchmarks. The five benchmarks are chosen because the provided services are common in modern smartphones and suffer from anomalies exhibited by Monkey. Each benchmark has several versions and we arbitrarily selected one. Brief descriptions of these benchmarks are listed below:

- AtPak (version 1.1.0) is a social-platform photo browser, which allows users to manage local photo galleries and upload photos to QQ.com.
- SMS Bomber (version 1.1) is an SMS text editor. Users can bomb a receiver by sending lots of messages in a short time.

- AnkiDroid (version 2.0 beta18) is a flashcard learning application. It helps users to manage vocabulary cards on the cloud and reminds them to review vocabulary in a desirable period. It also supports multiple languages and speech synthesis from texts.
- Surround (version 1.2) is a music player that allows for music sharing among nearby mobile devices.
- Taskcatapp (version 2011090101) is a memoir application which helps users to easily create, navigate, and search through memoirs.

8.2 Evaluation of Mined Temporal Properties

To measure the preciseness of the mined temporal properties, we also collect another 500 traces for each benchmark as the testing trace set to verify the accuracy of our miners. We designed a predicting mechanism to label these 500 traces as pass or fail according to the mined FLTL properties. If the prediction is with high accuracy, we can imply these mined properties are representive. Similar idea can be found in [9] which mined a classifier from execution traces to classify software behaviors.

The most difficult part of the predicting mechanism is to decide the importance of each property. In this work, for each cluster, we decide a set of weights(each one for a template) and a threshold by applying *cross-validation*[5] technique on the original 3000 traces. *Cross-validation* is a common technique in data mining for assessing how the result of statistical analysis will generalize to an independent data set.

By giving a new trace θ and the weights and the threshold of each cluster, the predicting mechanism works in the following procedures:

1. For each cluster, we sum up all the weights of properties which can be satisfied by θ(properties from the same template won't be count repeatedly). If the summation is higher than the threshold, then label θ as a potential member of the cluster.
2. If θ is not a potential member of any cluster, then θ is predicted as pass. Otherwise it is predicted as fail.

To measure the quality of our predicting mechanism, we compare the predicting result to the real execution result of these 500 traces and use 4 measures *accuracy*, *precision*, *recall* and *F-score* [16,17] to quantify the effectiveness of our implementation in test verdict prediction. To proceed, we first need the following concepts.

- A trace is a *true positive* if it is a 'fail' trace with test verdict correctly issued by our tool. Let *TP* be the number of true positives in the given trace set.
- A trace is a *true negative* if it is a 'pass' trace with test verdict correctly issued by our tool. labelled as fail. We let *TN* be the number of true negatives.
- A trace is a *false positive* (false alarm) if it is a 'pass' trace to which our tool issues a 'fail' verdict. We let *FP* be the number of false positives.
- A trace is a *false negative* if it is a 'fail' trace to which our tool issues a 'pass' verdict. Let *FN* be the number of false negatives.

Table 1. Accuracy of Clustered and no Clustered Mined Properties

	#Pass Traces	#Fail Traces	#Cluster Fail	TP	TN	FP	FN	Acc.	Prec.	Rec.	F-score
ATPak	458	42	1	5	433	25	37	87.6%	0.16	0.11	0.126
ATPak (Clustered)			2	38	424	34	4	92.4%	0.52	0.90	0.791
SMSBomber	473	27	1	22	401	72	5	84.6%	0.23	0.81	0.544
SMSBomber (Clustered)			2	27	418	55	0	89%	0.32	1	0.71
Taskcatapp	481	19	1	11	374	107	8	77%	0.09	0.57	0.283
Taskcatapp (Clustered)			2	13	453	28	6	93.2%	0.31	0.68	0.555
Surround	475	25	1	21	411	64	4	86.4%	0.24	0.84	0.567
Surround (Clustered)			3	20	451	24	5	94.2%	0.45	0.8	0.694
AnkiDroid	461	39	1	39	333	128	0	74.4%	0.23	1	0.603
AnkiDroid (Clustered)			3	33	461	0	6	98.8%	1	0.84	0.873

Table 2. Weights and Threshold Learned by Cross Validation

	Cluster Index	W1	W2	W3	W4	W5	Threshold
ATPak	1	1	1	1	1	6	8
	2	1	1	3	0	5	7
SMSBomber	1	1	0	0	1	8	1
	2	6	1	1	1	1	8
Taskcatapp	1	2	2	1	1	4	6
	2	3	0	1	5	1	7
Surround	1	3	1	4	1	1	8
	2	3	1	1	1	4	8
	3	3	2	2	1	2	8
AnkiDroid	1	1	2	2	1	4	7
	2	2	2	2	1	3	5
	3	3	2	2	1	2	8

Note that $Fscore_\beta$ is the harmonic mean of *Precision* and *Recall*, in our work we let $\beta = 2$ which means we weight *Recall* higher than *Precision*. We show the definitions of these metrics as follows:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \qquad Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP+FN} \qquad Fscore_\beta = (1+\beta\cdot\beta)\cdot\frac{Precision\cdot Recall}{\beta\cdot\beta\cdot Precision+Recall}$$

The performance of our mining algorithms is measured on these five benchmarks with clustered and unclustered fail traces. Table 1 shows the result while Table 2 shows the weights and the threshold learned by cross validation(and used in the prediction). The result shows that properties mined via clustered traces can predict with higher accuracy and precision to the new 500 traces than those via unclustered. This might show clustering can effectively reduce the interference among the behavior patterns of different negative clusters. Note that the weight in Table 2 shows that each template has been the most representitive one in some bugs(if we view each cluster represents an individual bug). This denotes the trigger behaviors of bugs are different and require different property templates to describe. When user want to find the root cause of a bug, he can start by viewing the properties with highest weight.

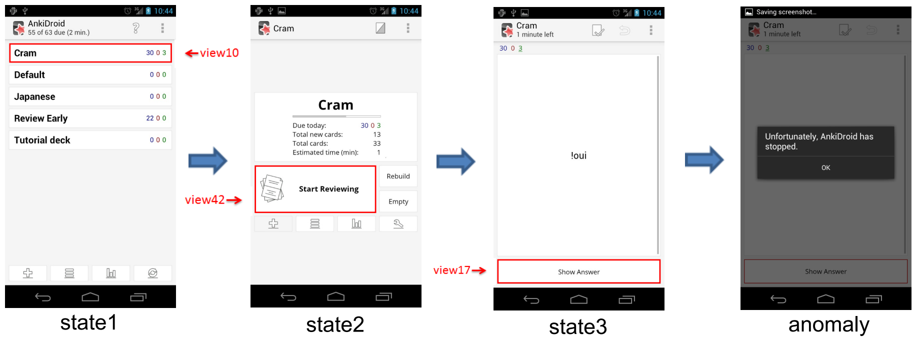

Fig. 3. State transition on AnkiDroid

8.3 Example of Mined Properties and Anomaly Analysis

We use one FLTL property mined via our tool to argue why our techniques can also be useful in diagnosis of anomalies exhibited in traces. The screen shots in Figure 3 demonstrate an anomaly existed in AnkiDroid. One Γ_2 property mined by our implementation is $G(state1 \rightarrow X(onTouch\_view10 \rightarrow X(state2 \rightarrow X(onTouch\_view42 \rightarrow X(state3 \rightarrow X(onTouch\_view17))))))$. This property shows an operation sequence leading to the exception with high confidence. Without this property, developers may only be aware of the existence of the anomaly but unaware of its reason. For example, bug reports from Google to an Android application developer only show the call stacks when a crash or freeze happens. Such a temporal property with high confidence could serve as a starting clue for root-causing the anomaly.

9 Conclusion and Future Works

In this work, we present an automated mining approach for detecting and diagnosing software defects. We have implemented our mining tool for Android applications. We chose five actively developing applications from Google Code and Google Play as benchmarks to test our approach. For diagnosis, we propose more types of temporal properties as mining target than previous approach did. To enhance the performance, we also propose to cluster the fail traces. By subtracting the mined temporal properties of pass traces from negative traces, the outcome temporal properties can effectively issue correct verdicts for each benchmark. The experiment data shows the effectiveness of our techniques.

In the future, we plan to mine more target types of temporal properties. Also, we expect to extend the framework from diagnosis to run-time monitoring and deploy our implementation on scalable industrial projects.

References

1. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD 1993, pp. 207–216. ACM, New York (1993)
2. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2002, pp. 4–16. ACM, New York (2002)
3. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA 2010, pp. 85–96. ACM, New York (2010)
4. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 213–224. ACM, New York (1999)
5. Geisser, S.: Predictive Inference: An Introduction. Monographs on Statistics and Applied Probability. Chapman & Hall (1993)
6. Havelund, K., Rosu, G.: Testing linear temporal logic formulae on finite execution traces. Technical report (2001)
7. Kamp, H.W.: Tense Logic and the Theory of Linear Order. Phd thesis, Computer Science Department, University of California at Los Angeles, USA (1968)
8. Kotsiantis, S., Kanellopoulos, D.: Association rules mining: A recent overview. GESTS International Transactions on Computer Science and Engineering 32(1), 71–82 (2006)
9. Lo, D., Cheng, H., Han, J., Khoo, S.-C., Sun, C.: Classification of software behaviors for failure detection: a discriminative pattern mining approach. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2009, pp. 557–566. ACM, New York (2009)
10. Lo, D., Khoo, S.-C.: Smartic: towards building an accurate, robust and scalable specification miner. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT 2006/FSE-14, pp. 265–275. ACM, New York (2006)
11. Lo, D., Khoo, S.-C., Liu, C.: Efficient mining of recurrent rules from a sequence database. In: Haritsa, J.R., Kotagiri, R., Pudi, V. (eds.) DASFAA 2008. LNCS, vol. 4947, pp. 67–83. Springer, Heidelberg (2008)
12. Lo, D., Khoo, S.-C., Liu, C.: Mining past-time temporal rules from execution traces. In: Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), WODA 2008, pp. 50–56. ACM, New York (2008)
13. Lo, D., Maoz, S., Khoo, S.-C.: Mining modal scenario-based specifications from execution traces of reactive systems. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE 2007, pp. 465–468. ACM, New York (2007)
14. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 501–510. ACM, New York (2008)
15. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS 1977, pp. 46–57. IEEE Computer Society, Washington, DC (1977)

16. Powers, D.M.W.: Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. Journal of Machine Learning Technologies 2(1), 37–63 (2011)
17. Rijsbergen, C.J.V.: Information Retrieval, 2nd edn. Butterworth-Heinemann, Newton (1979)
18. Strejček, J.: Linear Temporal Logic: Expressiveness and Model Checking. PhD thesis, Faculty of Informatics, Masaryk University, Brno, Czech Republic (2004)
19. Wang, F., Yao, L.-W., Wu, J.-H.: Intelligent test oracle construction for reactive systems without explicit specifications. In: Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing, DASC 2011.IEEE Computer Society, Washington, DC (2011)
20. Yang, J., Evans, D.: Perracotta: mining temporal api rules from imperfect traces. In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 282–291. ACM Press (2006)
21. Zhao, Q., Bhowmick, S.S.: Association Rule Mining: A Survey