Checking Activity Transition Systems with Back Transitions Against Assertions

Cunjing Ge[1,3], Jiwei Yan[2(✉)], Jun Yan[1,2,3], and Jian Zhang[1,3(✉)]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
{gecj,yanjun,zj}@ios.ac.cn
[2] Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China
yanjw@ios.ac.cn
[3] University of Chinese Academy of Sciences, Beijing, China

Abstract. The Android system is in widespread use currently, and Android apps play an important role in our daily life. How to specify and verify apps is a challenging problem. In this paper, we study a formalism for abstracting the behaviour of Android apps, called Activity Transition Systems (ATS), which includes back transitions, value assignments and assertions. Given such a transition system with a corresponding Activity Transition Graph (ATG), it is interesting to know whether it violates some value assertions. We first prove some theoretical properties of transitions and propose a state-merging strategy. Then we further introduce a post-reachability graph technique. Based on this technique, we design an algorithm to traverse an ATG that avoids path cycles. Lastly, we also extend our model and our algorithm to handle more complicated problems.

1 Introduction

The Android system, which provides rich and flexible features to ease the development of applications (apps), is one of the most popular mobile operating systems currently. Various Android apps are developed and released to the app market, which attracts high downloads due to the convenient interaction, user-friendly windows, and event-driven nature.

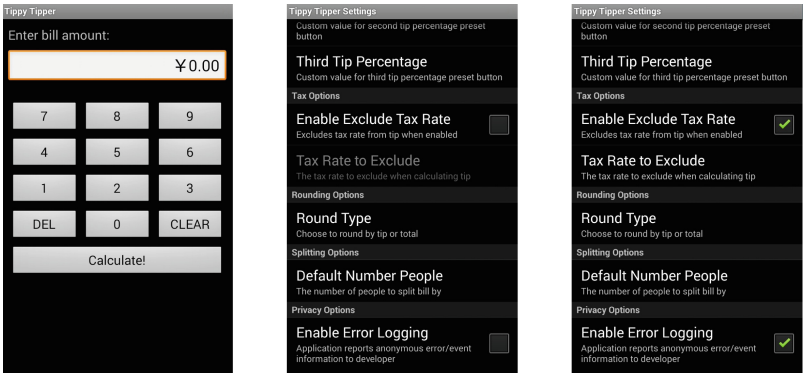
In Android system, the major component, **activity**, is a container which consists of various GUI widgets (e.g., button). Users can interact with widgets on an activity and trigger transitions between activities to perform a certain job. Thus activity transition model for event-driven callbacks is a fundamental model for analysis of Android apps. This serves as a cornerstone for many clients, such as vulnerability detection [6,9,12–15,19,20], malware detection and mitigation [10,11,19], GUI model generation [22,23], and GUI testing [3–5,16,17].

This work is partially supported by the National Key Basic Research (973) Program of China (Grant No.2014CB340701), the National Natural Science Foundation of China (Grant No. 61672505), and the Key Research Program of Frontier Sciences, CAS (Grant No. QYZDJ-SSW-JSC036).

J. Sun and M. Sun (Eds.): ICFEM 2018, LNCS 11232, pp. 388–403, 2018.
https://doi.org/10.1007/978-3-030-02450-5_23

All launched activities are arranged in the back stack in the order in which each activity is opened. Take a short message (SMS) manager app as an example, which may have an activity to show the list of contacts. When the user selects a contact person, a new activity is opened to view all the messages from or to the person. At the same time, the system will add the new activity to the back stack. Then if the user presses the back button on the bottom of the screen, that new activity is finished and popped off the stack. By default, activities in the stack can only be rearranged by push and pop operations. This back-stack mechanism is so flexible that a developer has to carefully inspect the status of the back stack when developing the transitions between activities. An activity with different back stacks may lead to different program behaviors, which brings the difficulty to the modeling of apps. When the launch-mode of activity is involved, the task will be more complicated.

Recent works [3,21,23,24] construct transition models of apps and traverse models to generate transition paths or even sequences to guide the GUI testing, some of them discuss the influences brought by the stack mechanism. These works adopt the same assumption that when the back operation is triggered, the model will roll back to the previous state. However, the assignments of global variables will not roll back. For example, the operations in the setting activity are also impossible to be rollback. As shown in Fig. 1, when the app `TippyTipper` are transited in the order of main $\xrightarrow{OpenSetting}$ setting1 $\xrightarrow{ClickCheckbox}$ setting2 $\xrightarrow{Back}$ main, the global variables that are changed in setting2 will not roll back by simply pressing the back button.



(a) main (b) setting1 (c) setting2

Fig. 1. Tippy tipper application

Because the back transition will lead to state change, in this paper, we consider a problem of determining if there exists a path that violates one of the assertions in the ATG with back transitions.

The main contributions of this work are summarized as follows:

- We propose an Activity Transition Graph (ATG) model with back transitions, value assignments and assertions, to describe the activity relations of Android apps in detail.
- We introduce a post-reachability graph and an algorithm to traverse an ATG that avoids path cycles.
- We extend our model and our algorithm to handle more complicated and also more interesting tasks.

The rest of this paper is organized as follows. Background and preliminary material is in Sect. 2, the algorithm in Sect. 3, several extensions of our model and approach in Sect. 4, related works in Sect. 5, and finally, concluding remarks in Sect. 6.

2 Background

Definition 1. *An **Activity Transition System (ATS)** $(X, \mathcal{V}, V_0, \mathcal{A}, A_0, \mathcal{T})$ consists of a set X of Boolean-valued variables, a set $\mathcal{V}$ of domains of variables in X, an initial assignment V_0, a set $\mathcal{A}$ of activities, an initial activity $A_0 \in \mathcal{A}$ and a set $\mathcal{T}$ of transitions.*

Each transition $\tau \in \mathcal{T}$ is a tuple (A, A') where A and A' are activities. Each activity or transition corresponds to a set of statements such as assignment statements like $x := 0$ and assertions like $x = 1 \rightarrow y = 0$.

Definition 2. *Given an ATS $(X, \mathcal{V}, V_0, \mathcal{A}, A_0, \mathcal{T})$, the **Activity Transition Graph (ATG)** is a digraph which is constructed in the following way:*

1. *For each activity in $\mathcal{A}$, introduce a vertex A_i.*
2. *For each transition $\tau = (A_i, A_j)$, introduce an edge from A_i to A_j.*

We introduce a special **back** transition τ_b which transits from the latest visited activity A_k to A_{k-1}. The statements of activity A_{k-1} will not be executed after back transition. Back transition not only rolls back activity, but also the part of assignments.

Definition 3. *A variable is **global** if it does not roll back its assignment during back transitions. Otherwise, it is a **local** variable.*

Assume X consists of n **global** variables $X^G = \{x_1^G, \ldots, x_n^G\}$ and m **local** variables $X^L = \{x_1^L, \ldots, x_m^L\}$. We use V^G and V^L to represent the assignments of X^G and X^L respectively. So back transition generates the $k+1$ step $\langle A_{k+1}, V_{k+1}\rangle$ that $V_{k+1}^G = V_k^G$, $V_{k+1}^L = V_{k-1}^L$ and $A_{k+1} = A_{k-1}$. We extend the ATS and ATG with such back transitions.

In this paper, we consider a problem of determining if there exists a path that violates one of the assertions in the ATG with back transitions.

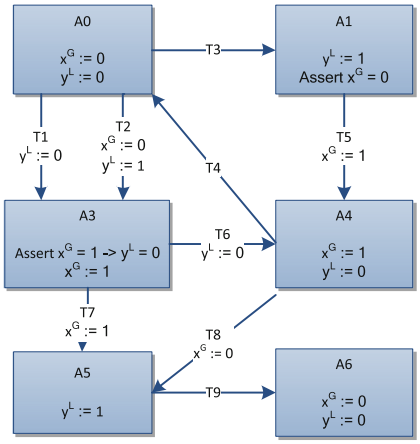


Fig. 2. An example of an ATG

Path:

$\langle A0,(0,0)\rangle \xrightarrow{T1} \langle A3,(1,0)\rangle \xrightarrow{back} \langle A0,(1,0)\rangle \xrightarrow{T3} \langle A1,(1,1)\rangle$

Statements:

```
A0     x := 0
A0     y := 0
T1     y := 0
A3     Assert x = 1 -> y = 0
A3     x := 1
BACK   y roll back 0
A1     y := 1
A1     Assert x = 0
```

Fig. 3. Statements of a path

Example. Figure 2 presents an example of the ATG with back transitions. It has 6 activities, 9 forward transitions and 2 boolean variables (one of them is global and another one is local). The initial values for variable x and y are both zero. Activity $A1$ and $A3$ contain assertions. The ATG starts from $A0$. There exists a path that violates the assertion in $A1$: $\langle A0,(0,0)\rangle \xrightarrow{T1} \langle A3,(1,0)\rangle \xrightarrow{back} \langle A0,(1,0)\rangle \xrightarrow{T3} \langle A1,(1,1)\rangle$. Concatenating the blocks of statements in transitions and activities, this path can be represented as Fig. 3. So a path can be considered as a sequence of statements. Since the initial assignment is determined, the assignment of variables on each statement is determined.

Relation with Pushdown Automata. There is a straightforward way to transform the assertion violation problem of an ATG with back transitions into a reachability problem of a pushdown automata:

- Let Q denote the set of states and Γ denote the stack alphabet. Since the assignments at an activity are finite, Q and Γ are also finite. We introduce an input symbol I_τ for the forward transition τ and an additional symbol $BACK$ to represent back transitions.
- Consider a pair (A,V) where A is an activity and V is an assignment. For (A,V), we introduce a state $q_{(A,V)} \in Q$ and a symbol $S_{(A,V)}$.
- We introduce a transition from $q_{(A_1,V_1)}$ to $q_{(A_2,V_2)}$ for a transition τ from activity A_1 to A_2 in the original ATG, where V_2 is the assignment result after executing statements of τ and A_2 with input V_1. This transition can be simulated by $(q_{(A_1,V_1)}, I_\tau, S^*, q_{(A_2,V_2)}, S_{(A_1,V_1)}S^*)$, where S^* represents an arbitrary stack symbol in Γ and $S_{(A_1,V_1)}S^*$ indicates that this transition pushes symbol $S_{(A_1,V_1)}$ into stack.
- Consider a back transition from (A_2,V_2) to the previous state (A_1,V_1) in the stack. Let V_0 represent the assignment after the back transition. We

first introduce two states $q_{(A_2,V_2)}$ and $q_{(A_1,V_0)}$. Then we introduce a transition $(q_{(A_2,V_2)}, BACK, S_{(A_1,V_1)}S^*, q_{(A_1,V_0)}, S^*)$ to simulate this back transition, where $S^* \in \Gamma$ and $S_{(A_1,V_1)}S^*$ indicates that this transition pops $S_{(A_1,V_1)}$.

3 Approach

We use a 3-tuple (A, V, S) to represent a state, where A is an activity, V is an assignment of variables and S is a stack that stores history information. Note that the stack S is a set of states which are previously visited, instead of a set of pairs like (A, V). Therefore, states containing different stacks are considered to be different in our model. A state contains necessary information for forward and back transitions. We can transit forward from one state to another and can also transit back to the previous state with the stack S.

3.1 A Straightforward Method

Algorithm 1 is the basic framework of breadth-first-search over the given ATG. It employs a queue Q to store states in this BFS algorithm. At the beginning, it adds the initial state (A_0, V_0, Φ) into Q. Then it visits every state in Q. For an unvisited element $q = (A, V, S)$ in Q, it enumerates each forward transition $\tau = (A, A_{next})$ from A. After that it executes statements and checks assertions on τ and the next activity A_{next} to obtain the new assignment V_{next}. After copy stack S to S_{next} and push state q into stack S_{next}, the algorithm adds the new state $(A_{next}, V_{next}, S_{next})$ into Q. After forward transitions, we consider the back transition at the state q. The algorithm pops the stack S to obtain the previous state $(A_{back}, V_{back}, S_{back})$. Since the assignment of global variables remains, we assign V^G to V^G_{back}. Then the algorithm adds the new state $(A_{back}, V_{back}, S_{back})$ into Q. At last, it visits another unvisited state in Q.

3.2 Post-reachability Graphs

The straightforward method may not terminate since it cannot handle cycle. Consider the example in Fig. 2, assume that we have already obtained a sequence of states: $7 : (A_0, (1,0), \Phi) \xrightarrow{T1} 15 : (A_3, (1,0), \{7\}) \xrightarrow{T6} S1 : (A_4, (1,0), \{7, 15\}) \xrightarrow{T4} S2 : (A_0, (0,0), \{7, 15, S1\})$. It is a sequence starting from state 7 to state $S2$ (states 7 and 15 are obtained in real execution of our algorithm while $S1$ and $S2$ are not, for details, see Fig. 6). Since the assignments of state 7 and state $S2$ are different, the sequence is not a cycle. However, if we start from state $S2$ through transition $T2$ and a back transition, we obtain a new state $S4 : (A_0, (1,0), \{7, 15, S1\})$. Then we find a cycle from state 7 to $S4$. The straightforward method will keep visiting activities starting from state $S4$, since $S4$ is different with state 7 in the perspective of stacks. In this example, it is also not sufficient to avoid cycles by only checking the existence of the pair $(A_0, (1,0))$, since it lacks path information. So, in this section, we introduce a

Algorithm 1: Straightforward Version

```
1  function
2      Q ← {(A_0, V_0, Φ)};
3      while Q not all visited do
4          pick an unvisited element q = (A, V, S) in Q;
5          for each τ = (A, A_next) start from A do
6              execute statements on τ and A_next and obtain V_next;
7              if assertions on τ or A_next violated then return false;
8              S_next ← S, S_next.push(q);
9              Q ← Q ∪ {(A_next, V_next, S_next)};
10         if S ≠ Φ then
11             (A_back, V_back, S_back) ← S.pop();
12             V^G_back ← V^G;
13             Q ← Q ∪ {(A_back, V_back, S_back)};
14         set q visited;
```

post-reachability graph for each activity to store sufficient history information for cycle avoidance.

Consider two states (A, V, S) and (A, V, S') on same activity A. They contain same variable assignment V, but different stacks S and S'. Intuitively, the forward transitions starting from these two states will lead to similar results, since in this case, stacks of history states only affect back transitions. So we could merge these two states into a virtual state with variable assignment V for the exploration of forward transitions. In other words, given a new state (A, V, S''), it is unnecessary to explore forward transitions starting from it. However, we have to store S, S' and S'' as they represent different path traces which are useful for the exploration of back transitions. To precisely describe the previous strategy, we introduce following lemmas and Theorem 1.

Lemma 1. *Given two states (A, V, S) and (A, U, R) on the same activity A. Consider a transition $\tau = (A, A_{next})$, let $(A_{next}, V_{next}, S_{next})$ and $(A_{next}, U_{next}, R_{next})$ denote the states after transition τ. Then $V = U \Rightarrow V_{next} = U_{next}$.*

Lemma 2. *Given two states (A, V, S) and (A, U, R) on the same activity A. Let $(A_{last}, V_{last}, S_{last})$ and $(A_{last}, U_{last}, R_{last})$ denote the last element of S and R respectively. Consider two states $(A_{last}, V_{back}, S_{last})$ and $(A_{last}, U_{back}, R_{last})$ after a back transition. Then $V = U, V_{last} = U_{last} \Rightarrow V_{back} = U_{back}$.*

Proof. Recall the definition of roll back operation on variable assignments, we know that $V^G_{back} = V^G$, $V^L_{back} = V^L_{last}$, $U^G_{back} = U^G$ and $U^L_{back} = U^L_{last}$. Since $V = U$ and $V_{last} = U_{last}$, it is obvious that $V^G = U^G$ and $V^L_{last} = U^L_{last}$. As a result, $V_{back} = U_{back}$. □

Theorem 1. *Given two states* (A_0, V_0, S_0) *and* (A_0, U_0, R_0) *on the same activity* A_0 *and a sequence of normal and back transitions* $\tau_1 \dots \tau_k$*. There are two sequences of states* $(A_0, V_0, S_0) \dots (A_k, V_k, S_k)$ *and* $(A_0, U_0, R_0) \dots (A_k, U_k, R_k)$*. Then* $\forall i \in \{1, \dots, k\}, V_0 = U_0, S_0 \subset S_i, R_0 \subset R_i \Rightarrow V_k = U_k$*.*

Proof. (Mathematical Induction)

Basis: τ_1 should be a forward transition as $S_0 \subset S_1$ and $R_0 \subset R_1$. From Lemma 1, we obtain $V_1 = U_1$ as $V_0 = U_0$.

Inductive Step: Show that $V_n = U_n$ if $V_0 = U_0, V_1 = U_1, \dots, V_{n-1} = U_{n-1}$.

Assume τ_n is a forward transition. From Lemma 1, we obtain $V_n = U_n$ as $V_{n-1} = U_{n-1}$. Assume τ_n is a back transition. We observe that stacks S_{n-1} and R_{n-1} are parts of sequences $(A_0, V_0, S_0) \dots (A_{n-2}, V_{n-2}, S_{n-2})$ and $(A_0, U_0, R_0) \dots (A_{n-2}, U_{n-2}, R_{n-2})$. So the last elements of S_{n-1} and R_{n-1} should be a pair of states $\big((A_l, V_l, S_l), (A_l, U_l, R_l)\big)$ from two sequences, where $0 \leq l \leq n-2$. From induction hypothesis, we know that $V_l = U_l$. From Lemma 2, we obtain $V_n = U_n$ as $V_{n-1} = U_{n-1}$ and $V_l = U_l$. □

Theorem 1 shows that two states with same variable assignments are always equivalent after a sequence of transitions (the number of forward transitions is not less than back transitions) in perspective of variable assignments. When the number of back transitions is more than forward transitions, we only have to consider the stacks of two states respectively. To apply such strategy in algorithm, we introduce the following concepts of post-reachable state and post-reachability graph.

Definition 4. *Given a state* (A_0, V_0, S_0)*. After a sequence of normal and back transitions* $\tau_1 \dots \tau_k$*, we obtain a sequence of states* $(A_0, V_0, S_0) \dots (A_k, V_k, S_k)$ *that* $\forall i \in \{1, \dots, k\}, S \subset S_i$*. If* $A_k = A_0$ *and* $S_k = S_0$*, the state* (A_k, V_k, S_k) *is called a* ***post-reachable state*** *of* (A_0, V_0, S_0)*.*

Definition 5. *Given a set of states* $\mathcal{S} = \{(A, V_1, S_1), \dots, (A, V_n, S_n)\}$ *on an activity* A*. Then the* ***Post-Reachabilitiy Graph (PRG)*** *over* $\mathcal{S}$ *is a digraph which is constructed in the following steps:*

1. *For each different variable assignment* V_i *in* $\mathcal{S}$*, introduce a vertex* v_i*.*
2. *For each vertex* v_i*, introduce the set of stacks* $\bigcup_{V_i = V_j} \{S_j\}$ *as its vertex value.*
3. *If* (A, V_j, S_j) *is a post-reachable state of* (A, V_i, S_i)*, where* $V_j \neq V_i$*, introduce an edge from* v_i *to* v_j*.*

In general, the PRG merges states with same variable assignment, and also stores different stacks for the exploration of back transitions. Figure 4 shows an example of a PRG over 8 states

$$\mathcal{S} = \{(A, (1,0,0), \{1\}), (A, (1,0,0), \{3\}), (A, (1,0,1), \{3\}), (A, (0,1,1), \{1,2\}), (A, (0,1,1), \{3\}), (A, (1,1,1), \{7\}), (A, (0,0,0), \{1,2\}), (A, (0,0,0), \{7\})\}.$$

There are 5 different value assignments in $\mathcal{S}$, so there are 5 vertices in this PRG. Each vertex corresponds to a set of stacks, e.g., vertex $(0,1,1)$ corresponds to

stacks $\Sigma_{(0,1,1)} = \{\{1,2\},\{3\}\}$. The edges present the post-reachability between vertices. Besides the state-merging feature, the PRG also has a propagation property. We present this property in the following theorem.

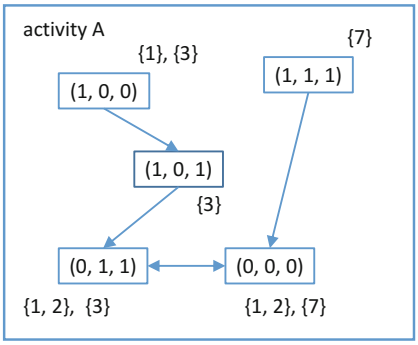


Fig. 4. An example of a PRG over 8 states

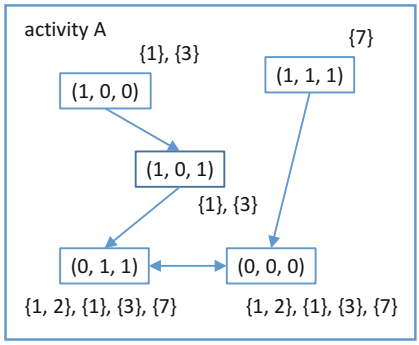


Fig. 5. Propagate values of vertices on the left PRG

Theorem 2. *Consider two vertices in a PRG of an activity A, e.g., v_1 and v_2. Let Σ_1 and Σ_2 denote the values of v_1 and v_2 (i.e., two sets of stacks), respectively. If there exists an edge from v_1 to v_2 in this PRG, we have $\Sigma_1 \subset \Sigma_2$.*

Proof. Since there exists an edge from v_1 to v_2, from the property of the post-reachable state, we could find a sequence of transitions that will transit (A, V_1, S) into (A, V_2, S), $\forall S \in \Sigma_1$. Then each stack in Σ_1 also belongs to Σ_2. □

Theorem 2 indicates that we could propagate values of vertices on an PRG. In the example of Fig. 4, we find that the stacks in $\Sigma_{(1,0,0)}$ should also belong to $\Sigma_{(1,0,1)}$, i.e., $\Sigma_{(1,0,1)} = \{\{1\},\{3\}\}$, from Theorem 2. As a result, we obtain the new PRG presented in Fig. 5 by such value propagation, which contains 5 more states.

3.3 The Algorithm with PRGs

Based on the PRG technique, we introduce our improved algorithm, which is also a BFS procedure. The pseudo-code is presented in Algorithm 2. It maintains a PRG G_A for each activity A. The value of the vertex V in G_A is denoted as $\Sigma_{G_A,V}$. Similar to the straightforward method, the improved BFS exploration also contains two parts: the exploration of forward transitions from line 5 to 11 and back transitions from line 12 to 15. Q is the queue of states to explore.

Algorithm 2 contains two sub-functions `InsertState()` and `AddEdge()`. In the function `InsertState`(A, V, S), we introduce a new vertex if V is different

Algorithm 2: Improved Version with PRGs

```
1  function
2      Q ← InsertState(A_0, V_0, Φ);
3      while Q not all visited do
4          pick an unvisited element q = (A, V, S) in Q;
5          if vertex V in G_A is not visited then
6              for each τ = (A, A_next) start from A do
7                  execute statements on τ and A_next and obtain V_next;
8                  if assertions on τ or A_next violated then return false;
9                  S_next ← S, S_next.push(q);
10                 Q ← Q ∪ InsertState(A_next, V_next, S_next);
11             set V in G_a visited;
12         if s ≠ Φ then
13             (A_back, V_last, S_back) ← S.pop();
14             V^L_back ← V^L_last, V^G_back ← V^G;
15             Q ← Q ∪ InsertState(A_back, V_back, S_back) ∪ AddEdge(A_back, V_last,
                V_back);
16         set q in Q visited;
17 function InsertState(A, V, S)
18     if V is not yet a vertex in G_A then
19         add vertex V into G_A and set Σ_{G_A,V} ← {S};
20     else
21         Σ_{G_A,V} ← Σ_{G_A,V} ∪ {S};
22     propagate on G_A and obtain new states S;
23     return {(A, V, S)} ∪ S;
24 function AddEdge(A, U, V)
25     add an edge ⟨U, V⟩ into G_A;
26     propagate on G_A and obtain new states S;
27     return S;
```

with the existing vertices in G_A, otherwise, we only have to update $\Sigma_{G_A,V}$ with the new stack S. Then we apply propagation procedure on G_A and return new states which are obtained in `InsertState()`. In the function `AddEdge`(A, U, V), we add a new edge from vertex U to vertex V ($U \neq V$) in G_A. Then it also propagates values on G_A and returns these new states.

The algorithm starts from the initial state (A_0, V_0, Φ). It invokes the function `InsertState()` to build the PRG with the initial state and generates the initial queue Q. Then the algorithm repeatedly enumerates unvisited states in Q.

For an unvisited state $q = (A, V, S)$, we first explore forward transitions starting from q. Recall the state-merging strategy over PRGs, we only have to explore the forward transition once for each variable assignment. So at line 5, the algorithm checks whether vertex V in G_A is already considered. Then it explores each forward transition $\tau = (A, A_{next})$. After that it executes statements and

checks assertions on τ and the next activity A_{next} to obtain the new assignment V_{next}. Then it copies stack S to S_{next} and pushes the state q into the stack S_{next}. At last, it obtains the new state $(A_{next}, V_{next}, S_{next})$ and invokes `InsertState()` for it.

After the forward transitions, we consider the back transition at state q. Note that different with the forward transitions, the back transition is always explored. At first, our approach pops S to obtain the previous state $(A_{back}, V_{last}, S_{back})$. From the definition of back transitions, we know that V^L_{back} is equal to V^L_{last} and V^G_{back} is equal to V^G. Thus our approach obtains the new state $(A_{back}, V_{back}, S_{back})$. Then it invokes `InsertState()` and `AddEdge()` to update G_A and Q. Recall the definition of post-reachable state that stacks of two states should be same, so there is no new edge during the exploration of forward transitions. However, since the algorithm has already explored a path from $(A_{back}, V_{last}, S_{back})$ to $(A_{back}, V_{back}, S_{back})$, the new state with V_{back} is the post-reachable state of the last state V_{last} when $V_{back} \neq V_{last}$. Thus `AddEdge()` is invoked at line 15. At last, the algorithm visits another unvisited state in Q.

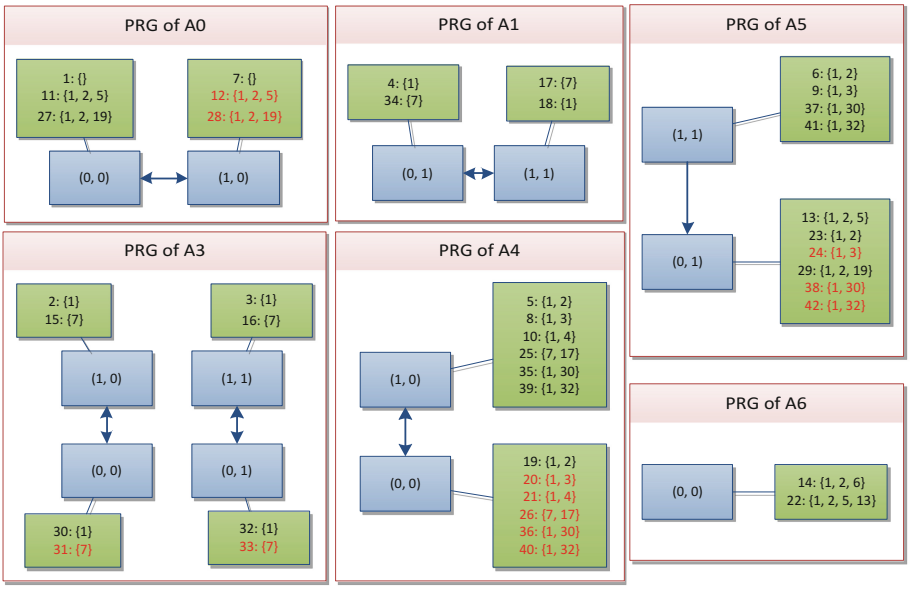


Fig. 6. All PRGs for activities in the example in Fig. 2 (Color figure online)

Example. Consider the cycle mentioned in Sect. 3.2: $7 : (A_0, (1,0), \Phi) \xrightarrow{T1} 15 : (A_3, (1,0), \{7\}) \xrightarrow{T6} S1 : (A_4, (1,0), \{7, 15\}) \xrightarrow{T4} S2 : (A_0, (0,0), \{7, 15, S1\}) \xrightarrow{T2} S3 : (A_3, (1,1), \{7, 15, S1, S2\}) \xrightarrow{back} S4 : (A_0, (1,0), \{7, 15, S1\})$. It is a cycle from A_0 to A_0. Since the vertex $(1,0)$ in G_{A_0} has already been visited at state

7, it is easy to see that our algorithm will not explore forward transitions from $S4$. In practice, the algorithm stops exploration of forward transitions earlier at $S2$, as the vertex $(0,0)$ in G_{A_0} has already been visited at the initial state $(A_0,(0,0),\Phi)$. Figure 6 presents the PRGs for all activities in the example in Fig. 2, which are generated by Algorithm 2. There are 42 states in total. Note that 12 of them (states in red) are generated by value propagation in PRGs.

4 Extensions

The model of ATG with back transitions is sometimes not sufficient for practical analysis over Android apps. However, our approach is flexible to extend to handle more complicated problems. In this section, we present several extensions to the ATG model and also our algorithm. Some extensions are orthogonal to each other, i.e., they could be employed at the same time.

4.1 Construct Paths for States

Our approach is designed for enumerating all possible different pairs of the activity and the variable assignment (A,V). Each (A,V) corresponds to a reachable state (A,V,S), i.e., there exists a path from the initial state to (A,V,S). Although Algorithm 2 guarantees that states are reachable, it does not store sufficient information to construct such path. Note that the path can be represented by a sequence of transitions. Thus we store the sequence of transitions T along with the stack, e.g., extend the state (A,V,S) to a 4-tuple (A,V,S,T).

For states obtained by forward and back transitions, it is simple to update T for this new state. But there are some states obtained by propagation in PRGs, whose sequences are not trivial to obtain. We introduce the following technique to handle such cases. First, for an edge $\langle V,V'\rangle$ in G_A, we store a sequence of transitions $T_{\langle V,V'\rangle}$ that will transit state (A,V,S) to (A,V',S) for all possible S. Then, when propagating (A,V,S,T) to (A,V',S,T') via edge $\langle V,V'\rangle$ in G_A, we could construct the new sequence T' by concatenating T with $T_{\langle V,V'\rangle}$.

For example, in PRG of A0 in Fig. 6, there is an edge $\langle(0,0)$ to $(1,0)\rangle$ and $T_{\langle(0,0),(1,0)\rangle}=\{T1,back\}$. Consider a state $(A0,(0,0),\{1,2,5\},\{T1,T6,T4\})$ at vertex $(0,0)$, where $\{T1,T6,T4\}$ is the sequence of transitions that forms a path from the initial state $(A0,(0,0),\Phi)$ to it. So we can obtain a new state $(A0,(1,0),\{1,2,5\},\{T1,T6,T4,T1,back\})$ by propagation.

4.2 Enumerated Variables and Arithmetic Expressions

We only consider Boolean-valued variables so far. However, it is easy to extend our approach to handle enumerated variables. For enumerated variables, our approach will still terminate in finite steps. Moreover, the results and techniques presented in previous sections will still work with this modification.

Since our approach only relies on sequential executions over statements instead of the satisfiability checking over constraints, it is easy to extend our

approach to support statements with complex expressions, such as, arithmetic expressions, comparison between variables, etc. In general, it supports extensions to expressions that return definite values after substituting values of variables. For example, $x := x + y$, assert $x > y$, $x := (y > 0)$.

4.3 Conditional Transitions

Transitions in the ATG do not contain any conditions. However, it is common that transitions between activities contain conditions in practical Android apps, e.g., transit from a Log-in activity to another unless users fill in correct passwords. For problems with such conditional transitions, it is sufficient to modify our approach by checking conditions before exploring transitions.

4.4 Self Loops

The self transition is a forward transition $\tau = (A, A)$. It also contains statements, but the statements of activity A will not be executed after τ. Note that the roll back operation is also enabled after τ. Self loops are generated by such self transitions. In practice, there are Android apps contain self loops. For example, in a video play activity, switching between the horizontal screen and the vertical screen is a self loop. Exploring a self transition is just like exploring a normal forward transition, except that the algorithm may have to introduce a new edge in a PRG during the exploration of a self transition. Based on this observation, we could extend our approach for self loops.

4.5 Overloading and Disabling Back Transitions

Overloading roll back function is common in practical Android apps, e.g., overload a back transition as a program exit. Moreover, overloading is so flexible that the overloaded transition may be very complicated. It may contain multiple functions. However, after some modifications, our approach still works. For example, in Fig. 7, the back transition on $A3$ is overloaded by a transition from $A3$ to $EXIT$. In this case, we have to disable the roll back operation at activity $A3$. Then we introduce a forward transition from $A3$ to $EXIT$ to simulate this overloading back transition.

Obviously, disabling a back transition can be viewed as a special case of overloading a back transition which does not introduce a new transition.

4.6 Activity Launch Modes

In Android, a parent activity can start a child activity by invoking, e.g., `startActivity()` as a form of an inter-component communication (ICC) call, passing it an intent that describes the child activity to be launched. In addition, an instance of an activity class A can be launched in one of the four launch modes, `standard`, `singleTask`, `singleTop` and `singleInstance`, either configured in

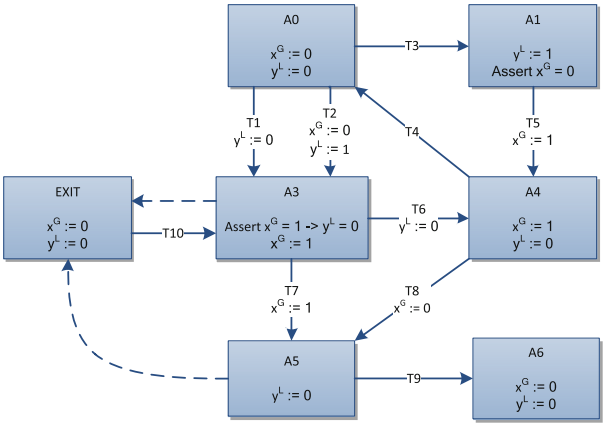


Fig. 7. An example of overloading back transitions

`AndroidManifest.xml` or specified in the intent passed to `startActivity()`. The first one is the default while the other three are known as special launch modes. These launch modes affect which activity instances are launched and their transitions.

standard. For the default launch mode, `standard` always creates a new activity instance of A and pushes the new instance into the back-stack. In our model, the default mechanism of forward transitions is exactly same as `standard`. Thus our algorithm naturally supports this mode.

singleTop. If the activity to be started has the same type as the top activity, then the top activity is reused. Otherwise, we handle it identically as in the case of `standard`. It is trivial to prove Theorem 1 for problems with `singleTop`. It shows that the state-merging strategy still works. However, the PRG technique is no longer working since forward transitions may pop element from stack. As a result, problems with this mode can be handled by Algorithm 1 with state-merging strategy, which also has termination guarantee.

singleTask. This mode is similar to `singleTop`, except that the activity instance closest to the top of the back-stack will be reused if it has the same type as the new activity to be started. Otherwise, we fall back to the case where `standard` is handled. For example, consider a forward transition $\tau = (A, A_{next})$ with `singleTask` and a state (A, V, S). Then we tries to find a state on activity A_{next} in S. If a state $(A_{next}, V_{old}, S_{old}) \in S$ is found, we adopt it as the next state and pops all state above it in S. Otherwise, we obtain the state $(A_{next}, V_{next}, S_{next})$ like `standard`. Similar to `singleTop`, the state-merging strategy still works, but the PRG technique is no longer working. Intuitively, transitions with `singleTask` will always generate states no more than `standard`. Therefore, Algorithm 1 with state-merging strategy also has termination guarantee.

singleInstance. This mode is similar to `singleTask`, except that only one instance of its activity class resides in its task. To simulate `singleInstance`, we have to maintain more than one stack for each state. Thus it is not trivial to extend our algorithm to this mode.

5 Related Work

An existing work [18] defines operational semantics for a fragment of Android that includes its Dalvik bytecode and intercommunication mechanism of the activities. It considers the Android specific activity stack and back operation. However, this work does not define GUI static models or give any analysis algorithms. Another work [7] proposes a formal model, Android Stack Machine (ASM), to capture key mechanisms of Android multi-tasking such as activities, back stacks, launch modes, as well as task affinities.

Aiming to describe real-world apps precisely, some static models are designed by researchers. Azim et al. [3] extract the Static activity Transfer Graph (SATG) for a given app, and use dynamic GUI exploration to handle dynamic activities layouts to complement the SATG. They also implement a tool A^3E which can explore real-world Android apps and construct models for them. S. Yang et al. [23] design a model called Window Transition Graph (WTG), with comprehensive behavior analysis for the key aspects of GUI behavior: widgets, event handlers, callback sequences, and especially the window stack changes. Based on the modeling of window stack, they develop analysis algorithms for WTG construction and traversal. And a recent work [24] constructs more precise activity Transition Graph with consideration of the launch-mode of each activity, which is more precise in capturing activity transitions. With help of the statically constructed activity Transition Model (ATM), Mirzaei et al. [17] give an approach to reduce the number of test cases by extracting the dependencies of GUI elements, which achieves a comparable coverage under exhaustive GUI testing using significantly fewer test cases.

Some researchers leverage dynamic techniques to construct transition model for Android apps. Amalfitano et al. [1,2] implemented a tool called *AnroidRipper* which builds model using a depth-first search over the user interface. When visiting a new state, it keeps a list of events belongs to the current state and systematically triggers them. And it restarts the exploration from the entry state when no new state can be detected in the current exploration. SwiftHand [8] builds an approximate model for the application under test, which could guide the test execution into unexplored parts of the state space while maximizing the code coverage and fault revelation. These works do not take into consideration the Android specific back stack. Yan et al. [21] make use of dynamic techniques to construct a labeled transition model (LATTE), which considers the information of activity back stack. They also implement a tool *LAND* to systematically explore real-world Android apps and construct the widget-sensitive and back-stack-aware models.

6 Conclusion

In this paper, an ATG with back transitions, value assignments and assertions, is introduced. It is a formalism for abstracting the behaviour of Android apps. Based on the PRG technique, we propose an algorithm for assertion checking over our formalism model, which has termination guarantee. Lastly, we study interesting extensions of our model and our algorithm. In the future, we would like to apply our algorithm to analyze Android apps with more activities and more states. On the other hand, automated modeling technique is also an interesting and challenging direction of our future works.

Acknowledgements. The authors are grateful to the reviewers for helpful comments and suggestions, and to Ping Wang for reading a preliminary version of this paper carefully.

References

1. Amalfitano, D., Fasolino, A.R., Tramontana, P., De Carmine, S., Memon, A.M.: Using GUI ripping for automated testing of Android applications. In: Proceedings of of ASE, pp. 258–261 (2012)
2. Amalfitano, D., Fasolino, A.R., Tramontana, P., Ta, B.D., Memon, A.M.: Mobiguitar: automated model-based testing of mobile apps. IEEE Softw. **32**(5), 53–59 (2015)
3. Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of Android apps. In: Proceedings of of OOPSLA, pp. 641–660 (2013)
4. Baek, Y.M., Bae, D.: Automated model-based android GUI testing using multi-level GUI comparison criteria. In: Proceedings of of ASE, pp. 238–249 (2016)
5. Bhoraskar, R., et al.: Brahmastra: driving apps to test the security of third-party components. In: Proceedings of USENIX, pp. 1021–1036 (2014)
6. Chen, Q.A., Qian, Z., Mao, Z.M.: Peeking into your app without actually seeing it: UI state inference and novel android attacks. In: Proceedings of USENIX, pp. 1037–1052 (2014)
7. Chen, T., He, J., Song, F., Wang, G., Wu, Z., Yan, J.: Android stack machine. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 487–504. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_29
8. Choi, W., Necula, G.C., Sen, K.: Guided GUI testing of android apps with minimal restart and approximate learning. In: Proceedigns of OOPSLA, pp. 623–640 (2013)
9. Do, L.N.Q., Ali, K., Livshits, B., Bodden, E., Smith, J., Murphy-Hill, E.R.: Just-in-time static analysis. In: Proceedings of SIGSOFT, pp. 307–317 (2017)
10. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: semantics-based detection of android malware through static analysis. In: Proceedings of FSE, pp. 576–587 (2014)
11. Feng, Y., Bastani, O., Martins, R., Dillig, I., Anand, S.: Automated synthesis of semantic malware signatures using maximum satisfiability. In: Proceedings of NDSS (2017)
12. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in DroidSafe. In: Proceedings of NDSS (2015)

13. Huang, W., Dong, Y., Milanova, A., Dolby, J.: Scalable and precise taint analysis for android. In: Proceedings of ISSTA, pp. 106–117 (2015)
14. Li, L., et al.: IccTA: detecting inter-component privacy leaks in android apps. In: Proceedings of ICSE, pp. 280–291 (2015)
15. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of CCS, pp. 229–240 (2012)
16. Mahmood, R., Mirzaei, N., Malek, S.: Evodroid: segmented evolutionary testing of android apps. In: Proceedings of FSE, pp. 599–609 (2014)
17. Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., Malek, S.: Reducing combinatorics in GUI testing of Android applications. In: Proceedings of ICSE, pp. 559–570 (2016)
18. Payet, E., Spoto, F.: An operational semantics for android activities. In: Proceedings of PEPM, pp. 121–132 (2014)
19. Shao, Y., Luo, X., Qian, C., Zhu, P., Zhang, L.: Towards a scalable resource-driven approach for detecting repackaged android applications. In: Proceedings of ACSAC, pp. 56–65 (2014)
20. Wei, F., Roy, S., Ou, X., Robby: Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of SIGSAC, pp. 1329–1341 (2014)
21. Yan, J., Wu, T., Yan, J., Zhang, J.: Widget-sensitive and back-stack-aware GUI exploration for testing android apps. In: Proceedings of QRS, pp. 42–53 (2017)
22. Yang, S., Yan, D., Wu, H., Wang, Y., Rountev, A.: Static control-flow analysis of user-driven callbacks in android applications. In: Proceedings of ICSE, pp. 89–99 (2015)
23. Yang, S., Zhang, H., Wu, H., Wang, Y., Yan, D., Rountev, A.: Static window transition graphs for Android(T). In: Proceedings of ASE, pp. 658–668 (2015)
24. Zhang, Y., Sui, Y., Xue, J.: Launch-mode-aware context-sensitive activity transition analysis. In: ICSE (2018, accepted)