Abstract. In this paper, we propose Android Stack Machine (ASM), a formal model to capture key mechanisms of Android multi-tasking such as activities, back stacks, launch modes, as well as task affinities. The model is based on pushdown systems with multiple stacks, and focuses on the evolution of the back stack of the Android system when interacting with activities carrying specific launch modes and task affinities. For formal analysis, we study the reachability problem of ASM. While the general problem is shown to be undecidable, we identify expressive fragments for which various verification techniques for pushdown systems or their extensions are harnessed to show decidability of the problem.

1 Introduction

Multi-tasking plays a central role in the Android platform. Its unique design, via activities and back stacks, greatly facilitates organizing user sessions through tasks, and provides rich features such as handy application switching, background app state maintenance, smooth task history navigation (using the “back” button), etc [16]. We refer the readers to Sect. 2 for an overview.

Android task management mechanism has substantially enhanced user experiences of the Android system and promoted personalized features in app design. However, the mechanism is also notoriously difficult to understand. As a witness, it constantly baffles app developers and has become a common topic of question-and-answer websites (for instance, [2]). Surprisingly, the Android multi-tasking
mechanism, despite its importance, has not been thoroughly studied before, let alone a formal treatment. This has impeded further developments of computer-aided (static) analysis and verification for Android apps, which are indispensable for vulnerability analysis (for example, detection of task hijacking [16]) and app performance enhancement (for example, estimation of energy consumption [8]).

This paper provides a formal model, i.e., Android Stack Machine (ASM), aiming to capture the key features of Android multi-tasking. ASM addresses the behavior of Android back stacks, a key component of the multi-tasking machinery, and their interplay with attributes of the activity. In this paper, for these attributes we consider four basic launch modes, i.e., standard (STD), singleTop (STP), singleTask (STK), singleInstance (SIT), and task affinities. (For simplicity more complicated activity attributes such as allowTaskReparenting will not be addressed in the present paper.) We believe that the semantics of ASM, specified as a transition system, captures faithfully the actual mechanism of Android systems. For each case of the semantics, we have created “diagnosis” apps with corresponding launch modes and task affinities, and carried out extensive experiments using these apps, ascertaining its conformance to the Android platform. (Details will be provided in Sect. 3.)

For Android, technically ASM can be viewed as the counterpart of pushdown systems with multiple stacks, which are the de facto model for (multi-threaded) concurrent programs. Being rigours, this model opens a door towards a formal account of Android’s multi-tasking mechanism, which would greatly facilitate developers’ understanding, freeing them from lengthy, ambiguous, elusive Android documentations. We remark that it is known that the evolution of Android back stacks could also be affected by the intent flags of the activities. ASM does not address intent flags explicitly. However, the effects of most intent flags (e.g., FLAG_ACTIVITY_NEW_TASK, FLAG_ACTIVITY_CLEAR_TOP) can be simulated by launch modes, so this is not a real limitation of ASM.

Based on ASM, we also make the first step towards a formal analysis of Android multi-tasking apps by investigating the reachability problem which is fundamental to all such analysis. ASM is akin to pushdown systems with multiple stacks, so it is perhaps not surprising that the problem is undecidable in general; in fact, we show undecidability for most interesting fragments even with just two launch modes. In the interest of seeking more expressive, practice-relevant decidable fragments, we identify a fragment STK-dominating ASM which assumes STK activities have different task affinities and which further restricts the use of SIT activities. This fragment covers a majority of open-source Android apps (e.g., from Github) we have found so far. One of our technical contributions is to give a decision procedure for the reachability problem of STK-dominating ASM, which combines a range of techniques from simulations by pushdown systems with transductions [19] to abstraction methods for multi-stacks. The work, apart from independent interests in the study of multi-stack pushdown systems, lays a solid foundation for further (static) analysis and verification of Android apps related to multi-tasking, enabling model checking of Android apps, security analysis (such as discovering task hijacking), or typical tasks in software engineering such as automatic debugging, model-based testing, etc.
We summarize the main contributions as follows: (1) We propose—to the best of our knowledge—the first comprehensive formal model, Android stack machine, for Android back stacks, which is also validated by extensive experiments. (2) We study the reachability problem for Android stack machine. Apart from strongest possible undecidability results in the general case, we provide a decision procedure for a practically relevant fragment.

2 Android Stack Machine: An Informal Overview

In Android, an application, usually referred to as an app, is regarded as a collection of activities. An activity is a type of app components, an instance of which provides a graphical user interface on screen and serves the entry point for interacting with the user [1]. An app typically has many activities for different user interactions (e.g., dialling phone numbers, reading contact lists, etc). A distinguished activity is the main activity, which is started when the app is launched. A task is a collection of activities that users interact with when performing a certain job. The activities in a task are arranged in a stack in the order in which each activity is opened. For example, an email app might have one activity to show a list of latest messages. When the user selects a message, a new activity opens to view that message. This new activity is pushed to the stack. If the user presses the “Back” button, an activity is finished and is popped off the stack. [In practice, the onBackPressed() method can be overloaded and triggered when the “Back” button is clicked. Here we assume—as a model abstraction—that the onBackPressed() method is not overloaded.] Furthermore, multiple tasks may run concurrently in the Android platform and the back stack stores all the tasks as a stack as well. In other words, it has a nested structure being a stack of stacks (tasks). We remark that in android, activities from different apps can stay in the same task, and activities from the same app can enter different tasks.

Typically, the evolution of the back stack is dependent mainly on two attributes of activities: launch modes and task affinities. All the activities of an app, as well as their attributes, including the launch modes and task affinities, are defined in the manifest file of the app. The launch mode of an activity decides the corresponding operation of the back stack when the activity is launched. As mentioned in Sect. 1, there are four basic launch modes in Android: “standard”, “singleTop”, “singleTask” and “singleInstance”. The task affinity of an activity indicates to which task the activity prefers to belong. By default, all the activities from the same app have the same affinity (i.e., all activities in the same app prefer to be in the same task). However, one can modify the default affinity of the activity. Activities defined in different apps can share a task affinity, or activities defined in the same app can be assigned with different task affinities. Below we will use a simple app to demonstrate the evolution of the back stack.

Example 1. In Fig. 1, an app ActivitiesLaunchDemo1 is illustrated. The app contains four activities of the launch modes STD, STP, STK and SIT, depicted

1 Adapted from an open-source app https://github.com/wauoen/LaunchModeDemo.
by green, blue, yellow and red, respectively. We will use the colours to name the activities. The green, blue and red activities have the same task affinity, while the yellow activity has a distinct one. The main activity of the app is the green activity. Each activity contains four buttons, i.e., the green, blue, yellow and red button. When a button is clicked, an instance of the activity with the colour starts. Moreover, the identifiers of all the tasks of the back stack, as well as their contents, are shown in the white zones of the window. We use the following execution trace to demonstrate how the back stack evolves according to the launch modes and the task affinities of the activities: The user clicks the buttons in the order of green, blue, blue, yellow, red, and green.

1. [Launch the app] When the app is launched, an instance of the main activity starts, and the back stack contains exactly one task, which contains exactly one green activity (see Fig. 1(a)). For convenience, this task is called the green task (with id: 23963).
2. [Start an STD activity] When the green button is clicked, since the launch mode of the green activity is STD, a new instance of the green activity starts and is pushed into the green task (see Fig. 1(b)).
3. [Start an STP activity] When the blue button is clicked, since the top activity of the green task is not the blue activity, a new instance of the blue activity is pushed into the green task (see Fig. 1(c)). On the other hand, if the blue button is clicked again, because the launch mode of the blue activity is STP and the top activity of the green task is already the blue one, a new instance of the blue activity will not be pushed into the green task and its content is kept unchanged.
4. [Start an STK activity] Suppose now that the yellow button is clicked, since the launch mode of the yellow activity is STK, and the task affinity of the yellow activity is different from that of the bottom activity of the green task, a new task is created and an instance of the yellow activity is pushed into the new task (called the yellow task, with id: 23964, see Fig. 1(d), where the leftmost task is the top task of the back stack).
5. [Start an SIT activity] Next, suppose that the red button is clicked, because the launch mode of the red activity is SIT, a new task is created and an instance of the red activity is pushed into the new task (called the red task, with id: 23965, see Fig. 1(e)). Moreover, at any future moment, the red activity is the only activity of the red task. Note that here a new task is created in spite of the affinity of the red activity.
6. [Start an STD activity from an SIT activity] Finally, suppose the green button is clicked again. Since the top task is the red task, which is supposed to contain only one activity (i.e., the red activity), the green task is then moved to the top of the back stack and a new instance of the green activity is pushed into the green task (see Fig. 1(f)).

3 Android Stack Machine

For $k \in \mathbb{N}$, let $[k] = \{1, \cdots, k\}$. For a function $f : X \rightarrow Y$, let $\text{dom}(f)$ and $\text{rng}(f)$ denote the domain ($X$) and range ($Y$) of $f$ respectively.
Definition 1 (Android stack machine). An Android stack machine (ASM) is a tuple $A = (Q, \text{Sig}, q_0, \Delta)$, where

- $Q$ is a finite set of control states, and $q_0 \in Q$ is the initial state,
- $\text{Sig} = (\text{Act}, \text{Lmd}, \text{Aft}, A_0)$ is the activity signature, where
  - $\text{Act}$ is a finite set of activities,
  - $\text{Lmd} : \text{Act} \rightarrow \{\text{STD}, \text{STP}, \text{STK}, \text{SIT}\}$ is the launch-mode function,
  - $\text{Aft} : \text{Act} \rightarrow [m]$ is the task-affinity function, where $m = |\text{Act}|$,
  - $A_0 \in \text{Act}$ is the main activity,
- $\Delta \subseteq Q \times (\text{Act} \cup \{\triangleright\}) \times \text{Inst} \times Q$ is the transition relation, where $\text{Inst} = \{\square, \text{back}\} \cup \{\text{start}(A) | A \in \text{Act}\}$, such that (1) for each transition $(q, A, \alpha, q') \in \Delta$, it holds that $q' \neq q_0$, and (2) for each transition $(q, \triangleright, \alpha, q') \in \Delta$, it holds that $q = q_0$, $\alpha = \text{start}(A_0)$, and $q' \neq q_0$.

For convenience, we usually write a transition $(q, A, \alpha, q') \in \Delta$ as $q \xrightarrow{A,\alpha} q'$, and $(q, \triangleright, \alpha, q') \in \Delta$ as $q \xrightarrow{\triangleright,\alpha} q'$. Intuitively, $\triangleright$ denotes an empty back stack, $\square$ denotes there is no change over the back stack, $\text{back}$ denotes the pop action, and $\text{start}(A)$ denotes the activity $A$ being started. We assume that, if the back stack is empty, the Android stack system terminates (i.e., no further continuation is possible) unless it is in the initial state $q_0$. We use $\text{Act}_{*}$ to denote $\{B \in \text{Act} | \text{Lmd}(B) = *\}$ for $* \in \{\text{STD}, \text{STP}, \text{STK}, \text{SIT}\}$.

Semantics. Let $A = (Q, \text{Sig}, q_0, \Delta)$ be an ASM with $\text{Sig} = (\text{Act}, \text{Lmd}, \text{Aft}, A_0)$.

A task of $A$ is encoded as a word $S = [A_1, \cdots, A_n] \in \text{Act}^+$ which denotes the content of the stack, with $A_1$ (resp. $A_n$) as the top (resp. bottom) symbol, denoted by $\text{top}(S)$ (resp. $\text{btm}(S)$). We also call the bottom activity of a non-empty task $S$ as the root activity of the task. (Intuitively, this is the first activity of the task.) For $* \in \{\text{STD}, \text{STP}, \text{STK}, \text{SIT}\}$, a task $S$ is called a $*$-task if $\text{Lmd}(\text{btm}(S)) = *$. We define the affinity of a task $S$, denoted by $\text{Aft}(S)$, to be $\text{Aft}(\text{btm}(S))$. For $S_1 \in \text{Act}^*$ and $S_2 \in \text{Act}^*$, we use $S_1 \cdot S_2$ to denote the concatenation of $S_1$ and $S_2$, and $\epsilon$ is used to denote the empty word in $\text{Act}^*$.

As mentioned in Sect. 2, the (running) tasks on Android are organized as the back stack, which is the main modelling object of ASM. Typically we write a back stack $\rho$ as a sequence of non-empty tasks, i.e., $\rho = (S_1, \cdots, S_n)$, where
S₁ and Sₙ are called the top and the bottom task respectively. (Intuitively, S₁ is the currently active task.) ε is used to denote the empty back stack. For a non-empty back stack ρ = (S₁, ⋯ , Sₙ), we overload top by using top(ρ) to refer to the task S₁, and thus top²(ρ) the top activity of S₁.

**Definition 2 (Configurations).** A configuration of A is a pair (q, ρ) where q ∈ Q and ρ is a back stack. Assume that ρ = (S₁, ⋯ , Sₙ) with Sᵢ = [Ai₁, ⋯ , Aiₘᵢ] for each i ∈ [n]. We require ρ to satisfy the following constraints:

1. For each Aᵢ ∈ Act_{STK} or Aᵢ ∈ Act_{SIT}, A occurs in at most one task. Moreover, if A occurs in a task, then A occurs at most once in that task. (At most one instance for each STK/SIT-task)
2. For each i ∈ [n] and j ∈ [mᵢ − 1] such that Aᵢ,j ∈ Act_{STP}, Aᵢ,j ≠ Aᵢ,j+1. [Non-stuttering for STP-activities]
3. For each i ∈ [n] and j ∈ [mᵢ] such that Aᵢ,j ∈ Act_{STK}, Aft(Aᵢ,j) = Aft(Sᵢ). [Affinities of STK-activities agree to the host task]
4. For each i ∈ [n] and j ∈ [mᵢ] such that Aᵢ,j ∈ Act_{SIT}, mᵢ = 1. [SIT-activities monopolize a task]
5. For i ≠ j ∈ [n] such that btm(Sᵢ) ∉ Act_{SIT} and btm(Sⱼ) ∉ Act_{SIT}, Aft(Sᵢ) ≠ Aft(Sⱼ). [Affinities of tasks are mutually distinct, except for those rooted at SIT-activities]

By Definition 2(5), each back stack ρ contains at most |Act_{SIT}| + |rng(Aft)| (more precisely, |Act_{SIT}| + |{Aft(A) | A ∈ Act \ Act_{SIT}}|) tasks. Moreover, by Definition 2(1–5), all the root activities in a configuration are pairwise distinct, which allows to refer to a task whose root activity is A as the A-task.

Let Confₐ denote the set of configurations of A. The initial configuration of A is (q₀, ε). To formalize the semantics of A concisely, we introduce the following shorthand stack operations and one auxiliary function. Here ρ = (S₁, ⋯ , Sₙ) is a non-empty back stack.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stack Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Noaction(ρ)</td>
<td>ρ</td>
</tr>
<tr>
<td>NewTask(B)</td>
<td>[B]</td>
</tr>
<tr>
<td>Push(ρ, B)</td>
<td>([B] ∙ S₁, S₂, ⋯ , Sₙ)</td>
</tr>
<tr>
<td>Pop(ρ)</td>
<td>ε, if n = 1 and S₁ = [A]; (S₂, ⋯ , Sₙ), if n &gt; 1 and S₁ = [A]; (S'₁, S₂, ⋯ , Sₙ), if S₁ = [A] ∙ S'₁ with S'₁ ∈ Act⁺;</td>
</tr>
<tr>
<td>PopUntil(ρ, B)</td>
<td>(S''₁, S₂, ⋯ , Sₙ), where S₁ = S'₁ ∙ S''₁ with S'₁ ∈ (Act \ {B})⁺ and top(S''₁) = B;</td>
</tr>
<tr>
<td>Move2Top(ρ, i)</td>
<td>(Sᵢ, S₁, ⋯ , Sᵢ₋₁, Sᵢ₊₁, ⋯ , Sₙ)</td>
</tr>
<tr>
<td>GetNonSITTaskByAft(ρ, k)</td>
<td>{ Sᵢ, if Aft(Sᵢ) = k and Lmd(btm(Sᵢ)) ≠ SIT; Undef, otherwise. }</td>
</tr>
</tbody>
</table>

Intuitively, GetNonSITTaskByAft(ρ, k) returns a non-SIT task whose affinity is k if it exists, otherwise returns Undef.

In the sequel, we define the transition relation (q, ρ) A→ (q′, ρ′) on Confₐ to formalize the semantics of A. We start with the transitions out of the initial state q₀ and those with □ or back action.
For each transition $q_0 \xrightarrow{A,\text{start}(A_0)} q, (q_0, \varepsilon) \xrightarrow{\text{A}} (q, \text{NewTask}(A_0))$.

- For each transition $q \xrightarrow{A, \square} q'$ and $(q, \rho) \in \text{Conf}_A$ such that $\text{top}^2(\rho) = A$, $(q, \rho) \xrightarrow{A} (q', \text{Noaction}(\rho))$.
- For each transition $q \xrightarrow{A, \text{back}} q'$ and $(q, \rho) \in \text{Conf}_A$ such that $\text{top}^2(\rho) = A$, $(q, \rho) \xrightarrow{A} (q', \text{Pop}(\rho))$.

The most interesting case is, however, the transitions of the form $q \xrightarrow{A, \text{start}(B)} q'$. We shall make case distinctions based on the launch mode of $B$. For each transition $q \xrightarrow{A, \text{start}(B)} q'$ and $(q, \rho) \in \text{Conf}_A$ such that $\text{top}^2(\rho) = A$, $(q, \rho) \xrightarrow{A} (q', \rho')$ if one of the following cases holds. Assume $\rho = (S_1, \cdots, S_n)$.

**Case Lmd($B$) = STD**

- Lmd($A$) \text{ SIT}, then $\rho' = \text{Push}(\rho, B)$;
- Lmd($A$) = SIT$^2$, then
  - if GetNonSITTaskByAft($\rho, \text{Aft}(B)) = S_i^3$, then $\rho' = \text{Push}(\text{Move2Top}(\rho, i), B)$,
  - if GetNonSITTaskByAft($\rho, \text{Aft}(B)) =$ Undef, then $\rho' = \text{NewTask}(\rho, B)$;

**Case Lmd($B$) = STP**

- Lmd($A$) \text{ SIT and } A \neq B, then $\rho' = \text{Push}(\rho, B)$;
- Lmd($A$) \text{ SIT and } A = B, then $\rho' = \text{Noaction}(\rho)$;
- Lmd($A$) = SIT (see footnote 2),
  - if GetNonSITTaskByAft($\rho, \text{Aft}(B)) = S_i$ (see footnote 3), then
    - * if top($S_i$) \text{ SIT, then } $\rho' = \text{Push}(\text{Move2Top}(\rho, i), B)$,
    - * if top($S_i$) = B, then $\rho' = \text{Move2Top}(\rho, i)$;
  - if GetNonSITTaskByAft($\rho, \text{Aft}(B)) =$ Undef, then $\rho' = \text{NewTask}(\rho, B)$;

**Case Lmd($B$) = SIT**

- A = B (see footnote 2), then $\rho' = \text{Noaction}(\rho)$;
- A \neq B and $S_i = [B]$ for some $i \in [n]^4$, then $\rho' = \text{Move2Top}(\rho, i)$;
- A \neq B and $S_i \neq [B]$ for each $i \in [n]$, then $\rho' = \text{NewTask}(\rho, B)$;

**Case Lmd($B$) = STK**

- Lmd($A$) \text{ SIT and } Aft(B) = Aft(S_1), then
  - if B does not occur in $S_i^5$, then $\rho' = \text{Push}(\rho, B)$;
  - if B occurs in $S_i^6$, then $\rho' = \text{PopUntil}(\rho, B)$;

- Lmd($A$) \text{ SIT } Aft(B) \neq Aft(S_1), then

---

2 By Definition 2(4), $S_1 = [A]$.
3 If $i$ exists, it must be unique by Definition 2(5). Moreover, $i > 1$, as Lmd($A$) = SIT.
4 If $i$ exists, it must be unique by Definition 2(1). Moreover, $i > 1$, as $A \neq B$.
5 B does not occur in $\rho$ at all by Definition 2(3–5).
6 Note that $B$ occurs at most once in $S_1$ by Definition 2(1).
• if GetNonSITTaskByAft(\(\rho, Aft(B)\)) = \(S_i\),
  * if \(B\) does not occur in \(S_i\) (see footnote 5), then \(\rho' = \text{Push}(\text{Move2Top}(\rho, i), B)\);
  * if \(B\) occurs in \(S_i\), then \(\rho' = \text{PopUntil}(\text{Move2Top}(\rho, i), B)\).

• if GetNonSITTaskByAft(\(\rho, Aft(B)\)) = \(\text{Undef}\), then \(\rho' = \text{NewTask}(\rho, B)\);

This concludes the definition of the transition definition of \(\rightarrow\). As usual, we use \(\xrightarrow{\rightarrow}\) to denote the reflexive and transitive closure of \(\rightarrow\).

**Example 2.** The ASM for the ActivitiesLaunchDemo app in Example 1 is \(\mathcal{A} = (Q, \text{Sig}, q_0, \Delta)\), where \(Q = \{q_0, q_1\}\), \(\text{Sig} = (\text{Act}, \text{Lmd}, \text{Aft}, A_g)\) with

- \(\text{Act} = \{A_g, A_b, A_y, A_r\}\), corresponding to the green, blue, yellow and red activity respectively in the ActivitiesLaunchDemo app,
- \(\text{Lmd}(A_g) = \text{STD}\), \(\text{Lmd}(A_b) = \text{STD}\), \(\text{Lmd}(A_y) = \text{STK}\), \(\text{Lmd}(A_r) = \text{SIT}\),
- \(\text{Aft}(A_g) = \text{Aft}(A_b) = \text{Aft}(A_r) = 1\), \(\text{Aft}(A_y) = 2\),

and \(\Delta\) comprises the transitions illustrated in Fig. 2. Below is a path in the graph \(\rightarrow\) corresponding to the sequence of user actions clicking the green, blue, yellow, red, blue button (cf. Example 1),

\[
(q_0, \varepsilon) \xrightarrow{\text{start}(A_g)} (q_1, ([A_g])) \xrightarrow{A_g, \text{start}(A_g)} (q_1, ([A_b, A_g])) \xrightarrow{A_b, \text{start}(A_b)} (q_1, ([A_b, A_g])) \xrightarrow{A_y, \text{start}(A_y)} (q_1, ([A_y], [A_b, A_g])) \xrightarrow{A_r, \text{start}(A_r)} (q_1, ([A_g, A_b, A_y, A_r], [A_y])).
\]

**Proposition 1.** Let \(\mathcal{A}\) be an ASM. For each \((q, \rho) \in \text{Conf}_{\mathcal{A}}\) and \((q, \rho) \xrightarrow{\rightarrow} (q', \rho')\), \((q', \rho') \in \text{Conf}_{\mathcal{A}}\), namely, \((q', \rho')\) satisfies the five constraints in Definition 2.

**Remark 1.** A single app can clearly be modeled by an ASM. However, ASM can also be used to model multiple apps which may share tasks/activities. (In this case, these multiple apps can be composed into a single app, where a new main activity is added.) This is especially useful when analysing, for instance, task hijacking [16]. We sometimes do not specify the main activity explicit for convenience. The translation from app source code to ASM is not trivial, but follows standard routines. In particular, in ASM, the symbols stored into the back stack are just

\[^7\] If \(i\) exists, it must be unique by Definition 2(5). Moreover, \(i > 1\), as \(\text{Lmd}(A) \neq \text{SIT} \implies \text{Aft}(B) \neq \text{Aft}(S_1)\).

\[^8\] Note that \(B\) occurs at most once in \(S_i\) by Definition 2(1).
names of activities. Android apps typically need to, similar to function calls of programs, store additional local state information. This can be dealt with by introducing an extend activity alphabet such that each symbol is of the form $A(b)$, where $A \in \text{Act}$ and $b$ represents local information. When we present examples, we also adopt this general syntax.

Model validation. We validate the ASM model by designing “diagnosis” Android apps with extensive experiments. For each case in the semantics of ASM, we design an app which contains activities with the corresponding launch modes and task affinities. To simulate the transition rules of the ASM, each activity contains some buttons, which, when clicked, will launch other activities. For instance, in the case of $\text{Lmd}(B) = \text{STD}$, $\text{Lmd}(A) = \text{SIT}$, $\text{GetNonSITTaskByAft}(\rho, \text{Aft}(B)) = \text{Undef}$, the app contains two activities $A$ and $B$ of launch modes SIT and STD respectively, where $A$ is the main activity. When the app is launched, an instance of $A$ is started. $A$ contains a button, which, when clicked, starts an instance of $B$. We carry out the experiment by clicking the button, monitoring the content of the back stack, and checking whether the content of the back stack conforms to the definition of the semantics. Specifically, we check that there are exactly two tasks in the back stack, one task comprising a single instance of $A$ and another task comprising a single instance of $B$, with the latter task on the top. Our experiments are done in a Redmi-4A mobile phone with Android version 6.0.1. The details of the experiments can be found at https://sites.google.com/site/assconformancetesting/.

4 Reachability of ASM

Towards formal (static) analysis and verification of Android apps, we study the fundamental reachability problem of ASM. Fix an ASM $A = (Q, \text{Sig}, q_0, \Delta)$ with $\text{Sig} = (\text{Act}, \text{Lmd}, \text{Aft}, A_0)$ and a target state $q \in Q$. There are usually two variants: the state reachability problem asks whether $(q_0, \varepsilon) \xrightarrow{A} (q, \rho)$ for some back stack $\rho$, and the configuration reachability problem asks whether $(q_0, \varepsilon) \xrightarrow{A} (q, \rho)$ when $\rho$ is also given. We show they are interchangeable as far as decidability is concerned.

**Proposition 2.** The configuration reachability problem and the state reachability problem of ASM are interreducible in exponential time.

Proposition 2 allows to focus on the state reachability problem in the rest of this paper. Observe that, when the activities in an ASM are of the same launch mode, the problem degenerates to that of standard pushdown systems or even finite-state systems. These systems are well-understood, and we refer to [6] for explanations. To proceed, we deal with the cases where there are exactly two launch modes, for which we have $\binom{4}{2} = 6$ possibilities. The classification is given in Theorems 1 and 2. Clearly, they entail that the reachability for general ASM (with at least two launch modes) is undecidable. To show the undecidability, we reduce from Minsky’s two-counter machines [14], which, albeit standard, reveals
the expressibility of ASM. We remark that the capability of \textit{swapping the order} of two distinct non-SIT-tasks in the back stack—\textit{without resetting} the content of any of them—is the main source of undecidability.

**Theorem 1.** The reachability problem of ASM is undecidable, even when the ASM contains only (1) STD and STK activities, or (2) STD and SIT activities, or (3) STK and STP activities, or (4) SIT and STP activities.

In contrast, we have some relatively straightforward positive results:

**Theorem 2.** The state reachability problem of ASM is decidable in polynomial time when the ASM contains STD and STP activities only, and in polynomial space when the ASM contains STK and SIT activities only.

As mentioned in Sect. 1, we aim to identify expressive fragments of ASM with decidable reachability problems. To this end, we introduce a fragment called STK-dominating ASM, which accommodates all four launch modes.

**Definition 3 (STK-dominating ASM).** An ASM is said to be STK-dominating if the following two constraints are satisfied:

1. The task affinities of the STK activities are mutually distinct,
2. For each transition $q \xrightarrow{A_{\text{start}}(B)} q' \in \Delta$ such that $A \in \text{Act}_{\text{SIT}}$, it holds that either $B \in \text{Act}_{\text{SIT}} \cup \text{Act}_{\text{STK}}$, or $B \in \text{Act}_{\text{STD}} \cup \text{Act}_{\text{STP}}$ and $\text{Aft}(B) = \text{Aft}(A_0)$.

The following result explains the name “STK-dominating”.

**Proposition 3.** Let $\mathcal{A} = (Q, \text{Sig}, q_0, \Delta)$ be an STK-dominating ASM with $\text{Sig} = (\text{Act}, \text{Lmd}, \text{Aft}, A_0)$. Then each configuration $(q, \rho)$ that is reachable from the initial configuration $(q_0, \varepsilon)$ in $\mathcal{A}$ satisfies the following constraints: (1) for each STK activity $A \in \text{Act}$ with $\text{Aft}(A) \neq \text{Aft}(A_0)$, $A$ can only occur at the bottom of some task in $\rho$, (2) $\rho$ contains at most one STD/STP-task, which, when it exists, has the same affinity as $A_0$.

It is not difficult to verify that the ASM given in Example 2 is STK-dominating.

**Theorem 3.** The state reachability of STK-dominating ASM is in 2-EXPTIME.

The proof of Theorem 3 is technically the most challenging part of this paper. We shall give a sketch in Sect. 5 with the full details in [6].

## 5 STK-dominating ASM

For simplicity, we assume that $\mathcal{A}$ contains STD and STK activities only. To tackle the (state) reachability problem for STK-dominating ASM, we consider two cases, i.e., $\text{Lmd}(A_0) = \text{STK}$ and $\text{Lmd}(A_0) \neq \text{STK}$. The former case is simpler

---

9 The more general case that $\mathcal{A}$ also contains STP and SIT activities is slightly more involved and requires more space to present, which can be found in [6].
because, by Proposition 3, all tasks will be rooted at STK activities. For the latter, more general case, the back stack may contain, apart from several tasks rooted at STK activities, one single task rooted at \( A_0 \). Sections 5.1 and 5.2 will handle these two cases respectively.

We will, however, first introduce some standard, but necessary, backgrounds on pushdown systems. We assume familiarity with standard finite-state automata (NFA) and finite-state transducers (FST). We emphasize that, in this paper, FST refers to a special class of finite-state transducers, namely, letter-to-letter finite-state transducers where the input and output alphabets are the same.

Preliminaries of Pushdown systems. A pushdown system (PDS) is a tuple \( \mathcal{P} = (Q, \Gamma, \Delta) \), where \( Q \) is a finite set of control states, \( \Gamma \) is a finite stack alphabet, and \( \Delta \subseteq Q \times \Gamma \times \Gamma^* \times Q \) is a finite set of transition rules. The size of \( \mathcal{P} \), denoted by \(|\mathcal{P}|\), is defined as \(|\Delta|\).

Let \( \mathcal{P} = (Q, \Gamma, \Delta) \) be a PDS. A configuration of \( \mathcal{P} \) is a pair \((q, w) \in Q \times \Gamma^*\), where \( w \) denotes the content of the stack (with the leftmost symbol being the top of the stack). Let \( \text{Conf}_\mathcal{P} \) denote the set of configurations of \( \mathcal{P} \). We define a binary relation \( \xrightarrow{\mathcal{P}} \) over \( \text{Conf}_\mathcal{P} \) as follows: \((q, w) \xrightarrow{\mathcal{P}} (q', w')\) iff \( w = \gamma w_1 \) and there exists \( w'' \in \Gamma^* \) such that \((q, \gamma, w'', q') \in \Delta \) and \( w' = w''w_1 \). We use \( \xrightarrow{\mathcal{P}} \) to denote the reflexive and transitive closure of \( \xrightarrow{\mathcal{P}} \).

A configuration \((q', w')\) is reachable from \((q, w)\) if \((q, w) \xrightarrow{\mathcal{P}} (q', w')\). For \( C \subseteq \text{Conf}_\mathcal{P} \), \( \text{pre}^*(C) \) (resp. \( \text{post}^*(C) \)) denotes the set of predecessor (resp. successor) reachable configurations \( \{(q', w') \mid \exists (q, w) \in C, (q', w') \xrightarrow{\mathcal{P}} (q, w)\} \) (resp. \( \{(q', w') \mid \exists (q, w) \in C, (q, w) \xrightarrow{\mathcal{P}} (q', w')\}\)). For \( q \in Q \), we define \( C_q = \{q\} \times \Gamma^* \) and write \( \text{pre}^*(q) \) and \( \text{post}^*(q) \) as shorthand of \( \text{pre}^*(C_q) \) and \( \text{post}^*(C_q) \) respectively.

As a standard machinery to solve reachability for PDS, a \( \mathcal{P}\)-multi-automaton (\( \mathcal{P}\)-MA) is an NFA \( \mathcal{A} = (Q', \Gamma, \delta, I, F) \) such that \( I \subseteq Q \subseteq Q' \) [4]. Evidently, \( \mathcal{P}\)-multi-automata are a special class of NFA. Let \( \mathcal{A} = (Q', \Gamma, \delta, I, F) \) be a \( \mathcal{P}\)-MA and \((q, w) \in \text{Conf}_\mathcal{P} \), \((q, w)\) is accepted by \( \mathcal{A} \) if \( q \in I \) and there is an accepting run \( q_0q_1 \cdots q_\ell \) of \( \mathcal{A} \) on \( w \) with \( q_0 = q \). Let \( \text{Conf}_\mathcal{A} \) denote the set of configurations accepted by \( \mathcal{A} \). Moreover, let \( \mathcal{L}(\mathcal{A}) \) denote the set of words \( w \) such that \((q, w) \in \text{Conf}_\mathcal{A} \) for some \( q \in I \). For brevity, we usually write \( \mathcal{A} \) instead of \( \mathcal{P}\)-MA when \( \mathcal{P} \) is clear from the context. Moreover, for an MA \( \mathcal{A} = (Q', \Gamma, \delta, I, F) \) and \( q' \in Q \), we use \( \mathcal{A}(q') \) to denote the MA obtained from \( \mathcal{A} \) by replacing \( I \) with \( \{q'\} \). A set of configurations \( C \subseteq \text{Conf}_\mathcal{P} \) is regular if there is an MA \( \mathcal{A} \) such that \( \text{Conf}_\mathcal{A} = C \).

Theorem 4 ([4]). Given a PDS \( \mathcal{P} \) and a set of configurations accepted by an MA \( \mathcal{A} \), we can compute, in polynomial time in \(|\mathcal{P}| + |\mathcal{A}|\), two MAs \( \mathcal{A}_{\text{pre}^*} \) and \( \mathcal{A}_{\text{post}^*} \) that recognise \( \text{pre}^*(\text{Conf}_\mathcal{A}) \) and \( \text{post}^*(\text{Conf}_\mathcal{A}) \) respectively.

The connection between ASM and PDS is rather obvious. In a nutshell, ASM can be considered as a PDS with multiple stacks, which is well-known to be undecidable in general. Our overall strategy to attack the state reachability problem for the fragments of ASM is to simulate them (in particular, the multiple stacks) via—in some cases, decidable extensions of—PDS.
5.1 Case $\text{Lmd}(A_0) = \text{STK}$

Our approach to tackle this case is to simulate $A$ by an extension of PDS, i.e., pushdown systems with transductions (TrPDS), proposed in [19]. In TrPDS, each transition is associated with an FST defining how the stack content is modified. Formally, a TrPDS is a tuple $P = (Q, \Gamma, \mathcal{T}, \Delta)$, where $Q$ and $\Gamma$ are precisely the same as those of PDS, $\mathcal{T}$ is a finite set of FSTs over the alphabet $\Gamma$, and $\Delta \subseteq Q \times \Gamma \times \Gamma^* \times \mathcal{T} \times Q$ is a finite set of transition rules. Let $\mathcal{R}(\mathcal{T})$ denote the set of transductions defined by FSTs from $\mathcal{T}$ and $[\mathcal{R}(\mathcal{T})]$ denote the closure of $\mathcal{R}(\mathcal{T})$ under composition and left-quotient. A TrPDS $P$ is said to be finite if $[\mathcal{R}(\mathcal{T})]$ is finite.

The configurations of $P$ are defined similarly as in PDS. We define a binary relation $\mathcal{R}$ on $\text{Conf}_P$ as follows: $(q, w) \mathcal{R} (q', w')$ if there are $\gamma \in \Gamma$, the words $w_1, u, w_2$, and $\mathcal{T} \in \mathcal{T}$ such that $w = \gamma w_1$, $(q, \gamma, u, \mathcal{T}, q') \in \Delta$, $w_1 \mathcal{T} \rightarrow w_2$, and $w' = uw_2$. Let $\mathcal{R}$ denote the reflexive and transitive closure of $\mathcal{R}$. Similarly to PDS, we can define $\text{pre}^*(\cdot)$ and $\text{post}^*(\cdot)$ respectively. Regular sets of configurations of TrPDS can be represented by MA, in line with PDS. More precisely, given a finite TrPDS $P = (Q, \Gamma, \mathcal{T}, \Delta)$ and an MA $A$ for $P$, one can compute, in time polynomial in $|P| + [\mathcal{R}(\mathcal{T})] + |A|$, two MAs $A_{\text{pre}^*}$ and $A_{\text{post}^*}$ that recognize the sets $\text{pre}^*(\text{Conf}_A)$ and $\text{post}^*(\text{Conf}_A)$ respectively [17–19].

To simulate $A$ via a finite TrPDS $P$, the back stack $\rho = (S_1, \cdots, S_n)$ of $A$ is encoded by a word $S_1\sharp \cdots \sharp S_n \bot$ (where $\sharp$ is a delimiter and $\bot$ is the bottom symbol of the stack), which is stored in the stack of $P$. Recall that, in this case, each task $S_i$ is rooted at an STK-activity which sits on the bottom of $S_i$. Suppose $\text{top}(S_1) = A$. When a transition $(q, A, \text{start}(B), q')$ with $B \in \text{Act}_{\text{STK}}$ is fired, according to the semantics of $A$, the $B$-task of $\rho$, say $S_i$, is switched to the top of $\rho$ and changed into $[B]$ (i.e., all the activities in the $B$-task, except $B$ itself, are popped). To simulate this in $P$, we replace every stack symbol in the place of $S_i$ with a dummy symbol $\dagger$ and keep the other symbols unchanged. On the other hand, to simulate a back action of $A$, $P$ continues popping until the next non-dummy and non-delimiter symbol is seen.

**Proposition 4.** Let $A = (Q, \text{Sig}, q_0, \Delta)$ be an STK-dominating ASM with $\text{Sig} = (\text{Act}, \text{Lmd}, \text{Aft}, A_0)$ and $\text{Lmd}(A_0) = \text{STK}$. Then a finite TrPDS $P = (Q', \Gamma, \mathcal{T}, \Delta')$ with $Q \subseteq Q'$ can be constructed in time polynomial in $|A|$ such that, for each $q \in Q$, $q$ is reachable from $(q_0, \varepsilon)$ in $A$ if and only if $q$ is reachable from $(q_0, \bot)$ in $P$.

For a state $q \in Q$, $\text{pre}^*_P(q)$ can be effectively computed as an MA $B_q$, and the reachability of $q$ in $A$ is reduced to checking whether $(q_0, \bot) \in \text{Conf}_{B_q}$.

5.2 Case $\text{Lmd}(A_0) \neq \text{STK}$

We then turn to the more general case $\text{Lmd}(A_0) \neq \text{STK}$ which is significantly more involved. For exposition purpose, we consider an ASM $A$ where there are exactly two STK activities $A_1, A_2$, and the task affinity of $A_2$ is the same as
that of the main task $A_0$ (and thus the task affinity of $A_1$ is different from that of $A_0$). We also assume that all the activities in $A$ are “standard” except $A_1, A_2$. Namely $\text{Act} = \text{Act}_{STD} \cup \{A_1, A_2\}$ and $A_0 \in \text{Act}_{STD}$ in particular. Neither of these two assumptions is fundamental and their generalization is given in [6].

By Proposition 3, there are at most two tasks in the back stack of $A$. The two tasks are either an $A_0$-task and an $A_1$-task, or an $A_2$-task and an $A_1$-task. An $A_2$-task can only surface when the original $A_0$-task is popped empty. If this happens, no $A_0$-task will be recreated again, and thus, according to the arguments in Sect. 5.1, we can simulate the ASM by TrPDS directly and we are done. The challenging case is that we have both an $A_0$-task and an $A_1$-task. To solve the state reachability problem, the main technical difficulty is that the order of the $A_0$-task and the $A_1$-task may be switched for arbitrarily many times before reaching the target state $q$. Readers may be wondering why they cannot simply simulate two-counter machines. The reason is that the two tasks are asymmetric in the sense that, each time when the $A_1$-task is switched from the bottom to the top (by starting the activity $A_1$), the content of the $A_1$-task is reset into $[A_1]$. But this is not the case for $A_0$-task: when the $A_0$-task is switched from the bottom to the top (by starting the activity $A_2$), if it does not contain $A_2$, then $A_2$ will be pushed into the $A_0$-task; otherwise all the activities above $A_2$ will be popped and $A_2$ becomes the top activity of the $A_0$-task. Our decision procedure below utilises the asymmetry of the two tasks.

Intuition of construction. The crux of reachability analysis is to construct a finite abstraction for the $A_1$-task and incorporate it into the control states of $A$, so we can reduce the state reachability of $A$ into that of a pushdown system $P_A$ (with a single stack). Observe that a run of $A$ can be seen as a sequence of task switching. In particular, an $A_0; A_1; A_0$ switching denotes a path in $\xrightarrow{A}$ where the $A_0$-task is on the top in the first and the last configuration, while the $A_1$-task is on the top in all the intermediate configurations. The main idea of the reduction is to simulate the $A_0; A_1; A_0$ switching by a “macro”-transition of $P_A$. Note that the $A_0$-task regains the top task in the last configuration either by starting the activity $A_2$ or by emptying the $A_1$-task. Suppose that, for an $A_0; A_1; A_0$ switching, in the first (resp. last) configuration, $q$ (resp. $q'$) is the control state and $\alpha$ (resp. $\beta$) is the finite abstraction of the $A_1$-task. Then for the “macro”-transition of $P_A$, the control state will be updated from $(q, \alpha)$ to $(q', \beta)$, and the stack content of $P_A$ is updated accordingly:

- If the $A_0$-task regains the top task by starting $A_2$, then the stack content is updated as follows: if the stack does not contain $A_2$, then $A_2$ will be pushed into the stack; otherwise all the symbols above $A_2$ will be popped.
- On the other hand, if the $A_0$-task regains the top task by emptying the $A_1$-task, then the stack content is not changed.

Roughly speaking, the abstraction of the $A_1$-task must carry the information that, when $A_0$-task and $A_1$-task are the top resp. bottom task of the back stack and $A_0$-task is emptied, whether the target state $q$ can be reached from the configuration at that time. As a result, we define the abstraction of the $A_1$-task
whose content is encoded by a word \( w \in \text{Act}^* \), denoted by \( \alpha(w) \), as the set of all states \( q'' \in Q \) such that the target state \( q \) can be reached from \( (q'', (w)) \) in \( A \). Note that during the process that \( q \) is reached from \( (q'', (w)) \) in \( A \), the \( A_0 \)-task does not exist anymore, but a (new) \( A_2 \)-task, may be formed. Let \( \text{Abs}_{A_1} = 2^Q \).

To facilitate the construction of the PDS \( P_{A} \), we also need to record how the abstraction “evolves”. For each \((q', A, \alpha) \in Q \times (\text{Act} \setminus \{A_1\}) \times \text{Abs}_{A_1} \), we compute the set \( \text{Reach}(q', A, \alpha) \) consisting of pairs \((q'', \beta)\) satisfying: there is an \( A_0; A_1; A_0 \) switching such that in the first configuration, \( A \) is the top symbol of the \( A_0 \)-task, \( q' \) (resp. \( q'' \)) is the control state of the first (resp. last) configuration, and \( \alpha \) (resp. \( \beta \)) is the abstraction for the \( A_1 \)-task in the first (resp. last) configuration.\(^{10}\)

### Computing \( \text{Reach}(q', A, \alpha) \). Let \((q', A, \alpha) \in Q \times (\text{Act} \setminus \{A_1\}) \times \text{Abs}_{A_1} \). We first simulate relevant parts of \( A \) as follows:

- Following Sect. 5.1, we construct a TrPDS \( P_{A} = (Q_{A}, I_{A}, T_{A}, \Delta_{A}) \) to simulate the \( A_1 \)-task and \( A_2 \)-task of \( A \) after the \( A_0 \)-task is emptied, where \( Q_{A} = Q \cup Q \times Q \) and \( I_{A} = \text{Act} \cup \{\sharp, \dagger\} \). Note that \( A_0 \) may still—as a “standard” activity—occur in \( P_{A} \) though the \( A_0 \)-task disappears.

In addition, we construct an MA \( B_w = (Q_{w}, I_{A}, \delta_{w}, I_{q}, F_{q}) \) to represent \( \text{pre}_{P_{A}}(q) \), where \( I_{q} \subseteq Q_{w} \). Then, given a stack content \( w \in \text{Act}_{ST} A_1 \) of the \( A_1 \)-task, the abstraction \( \alpha(w) \) of \( w \), is the set of \( q'' \in I_{q} \cap Q \) such that \( (q'', w \sharp \dagger) \in \text{Conf}_{B_w} \).

- We construct a PDS \( P_{A_0; A_2} = (Q_{A_0; A_2}, I_{A_0; A_2}, T_{A_0; A_2}, \Delta_{A_0; A_2}) \) to simulate the \( A_1 \)-task of \( A \), where \( I_{A_0; A_2} = (\text{Act} \setminus \{A_2\}) \cup \{\bot\} \). In addition, to compute \( \text{Reach}(q', A, \alpha) \) later, we construct an MA \( M_{(q', A, \alpha)} = (Q_{(q', A, \alpha)}, I_{A_0; A_2}, \delta_{(q', A, \alpha)}, I_{(q', A, \alpha)}, F_{(q', A, \alpha)}) \) to represent

\[
\text{post}_{P_{A_0; A_2}}^{+}((q_1, A_1 \bot) \mid (q', A, \text{start}(A_1), q_1) \in \Delta))\]

### Definition 4. \( \text{Reach}(q', A, \alpha) \) comprises

- the pairs \((q'', \beta) \in Q \times \text{Abs}_{A_1} \) satisfying that (1) \((q', A, \text{start}(A_1), q_1) \in \Delta \), (2) \((q_1, A_1 \bot) \xrightarrow[p_{A_0; A_2}]{\text{Abs}_{A_1}} (q_2, B w \bot) \), (3) \((q_2, B, \text{start}(A_2), q'') \in \Delta \), and (4) \( \beta \) is the abstraction of \( Bw \), for some \( B \in \text{Act} \setminus \{A_2\} \), \( w \in (\text{Act} \setminus \{A_2\})^* \) and \( q_1, q_2 \in Q \),

- the pairs \((q'', \bot) \) such that \((q', A, \text{start}(A_1), q_1) \in \Delta \) and \((q_1, A_1 \bot) \xrightarrow[p_{A_0; A_2}]{\text{Abs}_{A_1}} (q'', \bot) \) for some \( q_1 \in Q \).

Importantly, conditions in Definition 4 can be characterized algorithmically.

### Lemma 1. For \((q', A, \alpha) \in Q \times (\text{Act} \setminus \{A_1\}) \times \text{Abs}_{A_1} \), \( \text{Reach}(q', A, \alpha) \) is the union of

\[
\{(q'', \bot) \mid (q'', \bot) \in \text{Conf}_{M_{(q', A, \alpha)}} \}
\]

\(^{10}\) As we can see later, \( \text{Reach}(q', A, \alpha) \) does not depend on \( \alpha \) for the two-task special case considered here. We choose to keep \( \alpha \) in view of readability.
- the set of pairs $(q'', \beta) \in Q \times \text{Abs}_{A_1}$ such that there exist $q_2 \in Q$ and $B \in \text{Act} \setminus \{A_2\}$ satisfying that $(q_2, B, \text{start}(A_2), q'')$, and $(B(\text{Act} \setminus \{A_2\})^\ast \perp) \cap (\text{Act}_{\text{STD}}A_1)^\ast \perp) \cap (\mathcal{L}(\mathcal{M}'(q', A, \alpha))(q_2))^{(\perp)^{-1}} \perp \cap \mathcal{L}_{\beta} \neq \emptyset$, where $\mathcal{L}(\mathcal{M}'(q', A, \alpha))(q_2))^{(\perp)^{-1}}$ is the set of words $w$ such that $w \perp$ belongs to $\mathcal{L}(\mathcal{M}'(q', A, \alpha))(q_2))$, and $\mathcal{L}_\beta = \bigcap_{q'' \in \beta} \mathcal{L}(B_q(q'')) \cap \bigcap_{q'' \in Q \setminus \beta} \overline{\mathcal{L}(B_q(q''))}$, with $\overline{\mathcal{L}}$ representing the complement language of $\mathcal{L}$.

Construction of $\mathcal{P}_\mathcal{A}$. We first construct a PDS $\mathcal{P}_{A_0} = (Q_{A_0}, \Gamma_{A_0}, \Delta_{A_0})$, to simulate the $A_0$-task of $\mathcal{A}$. Here $Q_{A_0} = (Q \times \{0, 1\}) \cup (Q \times \{1\} \times \{\text{pop}\})$, $\Gamma_{A_0} = \text{Act}_{\text{STD}} \cup \{A_2, \perp\}$, and $\Delta_{A_0}$ comprises the transitions. Here 1 (resp. 0) marks that the activity $A_2$ is in the stack (resp. is not in the stack) and the tag \text{pop} marks that the PDS is in the process of popping until $A_2$. The construction of $\mathcal{P}_{A_0}$ is relatively straightforward, the details of which can be found in [6].

We then define the PDS $\mathcal{P}_\mathcal{A} = (Q_{\mathcal{A}}, \Gamma_{A_0}, \Delta_{\mathcal{A}})$, where $Q_{\mathcal{A}} = (\text{Abs}_{A_1} \times Q_{A_0}) \cup \{q\}$, and $\Delta_{\mathcal{A}}$ comprises the following transitions,

- for each $(p, \gamma, w, p') \in \Delta_{A_0}$ and $\alpha \in \text{Abs}_{A_1}$, we have $((\alpha, p), \gamma, w, (\alpha, p')) \in \Delta_{\mathcal{A}}$ (here $p, p' \in Q_{A_0}$, that is, of the form $(q', b)$ or $(q', b, \text{pop})$, [behaviour of the $A_0$-task]

- for each $(q', A, \alpha) \in Q \times (\text{Act} \setminus \{A_1\}) \times \text{Abs}_{A_1}$ and $b \in \{0, 1\}$ such that $\mathcal{M}(q', A, \alpha)(q) \neq \emptyset$, we have $((\alpha, (q', b)), A, A, q) \in \Delta_{\mathcal{A}}$, [switch to the $A_1$-task and reach $q$ before switching back]

- for each $(q', A, \alpha) \in Q \times (\text{Act} \setminus \{A_1\}) \times \text{Abs}_{A_1}$ and $(q'', \beta) \in \text{Reach}(q', A, \alpha)$ such that $\beta \neq \perp$,

  - if $A \neq A_2$, then we have $((\alpha, (q', 0)), A, A_2 A, (\beta, (q'', 1))) \in \Delta_{\mathcal{A}}$ and $((\alpha, (q', 1)), A, \varepsilon, (\beta, (q'', 1, \text{pop}))) \in \Delta_{\mathcal{A}}$,

  - if $A = A_2$, then we have $((\alpha, (q', 1)), A_2, A_2, (\beta, (q'', 1))) \in \Delta_{\mathcal{A}}$, [switch to the $A_1$-task and switch back to the $A_0$-task later by launching $A_2$]

- for each $(q', A, \alpha) \in Q \times (\text{Act} \setminus \{A_1\}) \times \text{Abs}_{A_1}$, $(q'', \perp) \in \text{Reach}(q', A, \alpha)$ and $b \in \{0, 1\}$, we have $((\alpha, (q', b)), A, (\emptyset, (q'', b))) \in \Delta_{\mathcal{A}}$, [switch to the $A_1$-task and switch back to the $A_0$-task later when the $A_1$-task becomes empty]

- for each $\alpha \in \text{Abs}_{A_1}$, $b \in \{0, 1\}$ and $A \in \text{Act}_{\text{STD}} \cup \{A_2\}$, $((\alpha, (q, b)), A, A, q) \in \Delta_{\mathcal{A}}$, [if $\emptyset$ is reached when the $A_0$-task is the top task]

- for each $q' \in Q$ and $\alpha \in \text{Abs}_{A_1}$ with $q' \in \alpha$, $((\alpha, (q', 0)), \perp, \perp, q) \in \Delta_{\mathcal{A}}$. [if $\emptyset$ is reached after the $A_0$-task becomes empty and the $A_1$-task becomes the top task]

**Proposition 5.** Let $\mathcal{A}$ be an STK-dominating ASM where there are exactly two STK-activities $A_1, A_2$ and $\text{Aft}(A_2) = \text{Aft}(A_0)$. Then $q$ is reachable from the initial configuration $(q_0, \varepsilon)$ in $\mathcal{A}$ iff $q$ is reachable from the initial configuration $((\emptyset, (q_0, 0)), \perp)$ in $\mathcal{P}_\mathcal{A}$. 
6 Related Work

We first discuss pushdown systems with multiple stacks (MPDSs) which are the most relevant to ASM. (For space reasons we will skip results on general pushdown systems though.) A multitude of classes of MPDSs have been considered, mostly as a model for concurrent recursive programs. In general, an ASM can be encoded as an MPDS. However, this view is hardly profitable as general MPDSs are obviously Turing-complete, leaving the reachability problem undecidable.

To regain decidability at least for reachability, several subclasses of MPDSs were proposed in literature: (1) bounding the number of context-switches [15], or more generally, phases [10], scopes [11], or budgets [3]; (2) imposing a linear ordering on stacks and pop operations being reserved to the first non-empty stack [5]; (3) restricting control states (e.g., weak MPDSs [7]). However, our decidable subclasses of ASM admit none of the above bounded conditions. A unified and generalized criterion [12] based on MSO over graphs of bounded tree-width was proposed to show the decidability of the emptiness problem for several restricted classes of automata with auxiliary storage, including MPDSs, automata with queues, or a mix of them. Since ASMs work in a way fairly different from multi-stack models in the literature, it is unclear—literally for us—to obtain the decidability by using bounded tree-width approach. Moreover, [12] only provides decidability proofs, but without complexity upper bounds. Our decision procedure is based on symbolic approaches for pushdown systems, which provides complexity upper bounds and which is amenable to implementation.

Higher-order pushdown systems represent another type of generalization of pushdown systems through higher-order stacks, i.e., a nested “stack of stack” structure [13], with decidable reachability problems [9]. Despite apparent resemblance, the back stack of ASM can not be simulated by an order-2 pushdown system. The reason is that the order between tasks in a back stack may be dynamically changed, which is not supported by order-2 pushdown systems.

On a different line, there are some models which have addressed, for instance, GUI activities of Android apps. Window transition graphs were proposed for representing the possible GUI activity (window) sequences and their associated events and callbacks, which can capture how the events and callbacks modify the back stack [21]. However, the key mechanisms of back stacks (launch modes and task affinities) were not covered in this model. Moreover, the reachability problem for this model was not investigated. A similar model, labeled transition graph with stack and widget (LATTE [20]) considered the effects of launch modes on the back stacks, but not task affinities. LATTE is essentially a finite-state abstraction of the back stack. However, to faithfully capture the launch modes and task affinities, one needs an infinite-state system, as we have studied here.

7 Conclusion

In this paper, we have introduced Android stack machine to formalize the back stack system of the Android platform. We have also investigated the decidability
of the reachability problem of ASM. While the reachability problem of ASM is undecidable in general, we have identified a fragment, i.e., STK-dominating ASM, which is expressive and admits decision procedures for reachability.

The implementation of the decision procedures is in progress. We also plan to consider other features of Android back stack systems, e.g., the “allowTaskReparenting” attribute of activities. A long-term program is to develop an efficient and scalable formal analysis and verification framework for Android apps, towards which the work reported in this paper is the first cornerstone.

References


Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.