Formalization of Android Activity-Fragment Multitasking Mechanism and Static Analysis of Mobile Apps

JINLONG HE, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China and School of Computer Science and Technology, University of Chinese Academy of Sciences, China

ZHILIN WU, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China and School of Computer Science and Technology, University of Chinese Academy of Sciences, China

TAOLUE CHEN, School of Computing and Mathematical Sciences, Birkbeck, University of London, UK

The multitasking mechanism between activities and fragments plays a fundamental role in the Android operating system, which involves a wide range of features, including launch modes, intent flags, task affinities, and structured activities containing fragments. All of them are being widely used in Android apps, both open-source and commercial ones. In this paper, we present a formal semantics of the Android multitasking mechanism between activities and fragments, which accommodates all the important features and gives insofar the most comprehensive and accurate formalization. In particular, our semantics is formulated based on multi-stack systems, and fully captures the behavior of task stacks and activity stacks regarding fragments. Based on the semantics, we provide new static analysis algorithms, which are both multi-stack-aware and fragment-sensitive, thus achieve more precise static analysis for Android apps. We validate our approach by extensive experiments on both open-source and commercial Android apps. The results highlight the benefits of the considering the semantics of the multitasking mechanism between activities and fragments in static analysis, and confirm the efficacy of our approach.

CCS Concepts: • **Theory of computation → Operational semantics**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: Android multitasking mechanism, activities and fragments, formal semantics, stack unboundedness, static analysis

ACM Reference Format:

Jinlong He, Zhilin Wu, and Taolue Chen. 2022. Formalization of Android Activity-Fragment Multitasking Mechanism and Static Analysis of Mobile Apps. *Form. Asp. Comput.* 1, 1, Article 1 (January 2022), 92 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' addresses: Jinlong He, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, No. 4 South 4th Street, Zhongguancun, Haidian District, Beijing, China and School of Computer Science and Technology, University of Chinese Academy of Sciences, No.19(A) Yuquan Road, Shijingshan District, Beijing, China, hejl@ios.ac.cn; Zhilin Wu, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, No. 4 South 4th Street, Zhongguancun, Haidian District, Beijing, China and School of Computer Science and Technology, University of Chinese Academy of Sciences, No.19(A) Yuquan Road, Shijingshan District, Beijing, China; Taolue Chen, School of Computing and Mathematical Sciences, Birkbeck, University of London, Malet Street, Bloomsbury, London, UK.

Manuscript submitted to ACM

1 INTRODUCTION

Android, a mobile operating system developed by Google, serves over 2 billion monthly active users and occupies over 80% of the share of the global mobile operating system market.[1] The Google Play App store, Google's official pre-installed app store on Android devices, has supplied 2 million apps since 2016.[2]

Multitasking mechanism between activities and fragments plays a fundamental role in the Android operating system. Mobile apps provide diverse functionalities with seamless user experience by interacting with other apps. For example, the Gallery app utilizes the Email app to send pictures rather than implements its own email functionality [3]. These features require the support of frequent switch between apps. Moreover, Android provides a "back" button by which users can easily switch between apps.

To facilitate smooth app switching and history recalling, Android introduces a last-in-first-out data structure, generally referred to as *back stack*, to store users' histories. Technically speaking, the back stack exhibits a three-tier nested structure. At the highest layer there is a stack of tasks where each task can be considered as a stack of activities (the middle layer), and each activity may contain fragment stacks (the lowest layer). In other words, the back stack exhibits a hierarchy of task stack, activity stack and fragment stack. From the functionality perspective, a task is a collection of activities that users interact with when performing a certain job (e.g., view email); an activity is a type of app components, which provides a graphical user interface (GUI) on screen and serves the entry point of the interaction with users. To facilitate GUI reuse, an activity may be further divided into fragments. A fragment defines and manages its own layout, has its own lifecycle, and can handle its own input events. It cannot live on its own, i.e., must be hosted by an activity.

Activities are a central concept in the Android multitasking mechanism. In particular, the evolution of the back stack depends on various attributes of activities, including launch modes and task affinities, as well as the intent flags when starting activities. Moreover, the updates of the fragment containers in activities are controlled by fragment transactions (cf. Section 2 for more details).

In light of its central role in analyzing Android apps, the multitasking mechanism between activities and fragments has been considered in literature. First, there have been decent work towards its formalization and we refer to readers to Section 1.1 for some introduction. However, the existing formal semantics is far from complete. A primary drawback is that the activities are treated as an atomic object therein, while the internal structures (e.g., fragments) were abstracted away. The understanding of semantics may play a significant role in carrying out semantics-aware analysis of the mobile app, which can capture more features of the Android platform and thus is more precise. In contrast, the state-of-the-art analysis largely applied a syntactic approach. The underlying stack semantics of the multitasking mechanism between activities and fragments is either ignored completely ([AN13, LWX18, CHGD18]) or considered in a very much restricted form (e.g., single stack [YZW+15]), or otherwise important features (e.g., task affinities or fragments) are skipped [YZW+15, YWYZ17, ZSX18].

The status quo is clearly unsatisfying. Indeed, all the multitasking features, including task affinities and fragments, are widely used in Android apps, especially the commercial ones. A small-scale survey on the 3,000 most downloaded commercial Android apps in Google play shows that almost all apps use launch modes and intent flags, about 10% of the apps use non-default task affinities, and about 50% of the apps involve fragments. We address these shortcomings in the current paper.

[1]https://expandedramblings.com/index.php/android-statistics/
[2]https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/
[3]https://play.google.com/store/apps/details?id=com.google.android.apps.photosgo&hl=en_US

Main contributions. In this paper, we make the following contributions.

- We introduce a new model, Android multitasking stack machine (AMASS), to define the formal semantics of the multitasking mechanism between activities and fragments. AMASS covers all the important multitasking features (e.g., task affinities and fragments) and provides, to the best of our knowledge, insofar the most comprehensive formal account of the Android multitasking mechanism between activities and fragments. Moreover, We validate the formal semantics of AMASS models against the actual behavior of Android apps by designing a special-purpose app and doing large-scale experiments with the app.
- We harness the semantics to design novel algorithms for static analysis of the abnormal behaviors of task unboundedness and fragment-container unboundedness.
- We implement the algorithms into a static analysis tool called TaskDroid and experiment on a corpus of open source and commercial Android apps. The results show that TaskDroid can detect the task/fragment-container unboundedness problems for open source and commercial Android apps and the unboundedness problems entail abnormal behaviors of Android apps, including black screen, app crash, and device reboot, owing to a fine-grained semantic modeling of Android multitasking mechanism between activities and fragments.

1.1 Related work

Formalization of Android multitasking semantics. In light of its central role in analyzing Android apps, the multitasking mechanism between activities and fragments has been considered in literature. To the best of our knowledge, this line of work was initiated in [AN13], where the concept of activity transition graphs (ATG) was proposed. ATGs are directed graphs where nodes are activities and edges describe the relationships that an activity can start the other. ATGs can be used for generating traces in GUI testing. An extension of ATGs, i.e., window transition graphs (WTGs) were proposed in [YZW+15]. In WTGs, windows are assumed to be stored in one stack; nodes are windows that include not only activities, but also menus and dialogs; transitions represent a sequence of push/pop operations associated with sequences of callbacks. ATGs were also extended in [YWYZ17], giving rise to the LATTE model, a finite-state machine model which stores the activities in a bounded-height stack as well as some state information transferred across activities. Moreover, in [ZSX18], activities were assumed to be stored into a stack and its contents were used as contexts for GUI testing and pointer analysis. In [LWX18], a tool called TDroid was developed to utilize the syntactic pattens on the launch modes, intent flags, and task affinities to detect the app switching attacks. All of the above work did not consider fragments which were addressed in [CHGD18], where activity fragment transition graphs (AFTG) were proposed and used for GUI testing. Furthermore, in [LHR17, CHS+18, HCW+19], formal semantics for Android multitasking mechanism were proposed.

One serious drawback of the current formalization is that the activities are treated as an atomic object therein, while the internal structures (e.g., fragments) were abstracted away. In this paper, we provide the first complete formal account of the operational semantics, differentiating different versions of Android, and carry out validation ensuring its conformance to the actual behavior.

Static analysis. Static analysis approaches have been applied to Android apps, where typical tasks include assessing the security of Android apps, detecting app clones, automating test cases generation, or uncovering non-functional issues related to performance or energy [LBP+17]. There is a large body of work where in general the analysis techniques are still those which have been thoroughly considered in the static analysis of programs, e.g., control-flow analysis, data-flow analysis, and point-to analysis, but are adapted to Android apps.

A specific purpose of static analysis which is the closest to this paper is to reveal security vulnerabilities, many of which are related to the ICC mechanism and its potential misuses such as component hijacking. For instance, [LLW+12] applied reachability analysis on customized system dependence graphs to detect component hijacking vulnerabilities, and [OMJ+13, OLD+15] implemented the detection of inter-component vulnerabilities. We refer to [LBP+17] for a systematic literature review for the static analysis of Android apps.

In terms of static analysis with respect to the multitasking mechanism between activities and fragments, the state-of-the-art analysis largely applied a syntactic approach. The underlying stack semantics of the multitasking mechanism between activities and fragments is either ignored completely ([AN13, LWX18, CHGD18]) or considered in a restricted form (e.g., single stack [YZW+15]), or otherwise important features (e.g., task affinities or fragments) are skipped [YZW+15, YWYZ17, ZSX18].

[LHR17] develops a static analysis tool based on the operational semantics formalized therein, which can analyze Android apps. The tool extracts the necessary information (e.g., activity, intent flag) and analyzes possible activity injection cases to detect activity injection attacks. We are however not aware of other formal approaches towards analysis of Android apps accounting for the multitasking mechanism between activities and fragments similar to what we present in the current paper.

Structure. The rest of the paper is organized as follows. Section 2 gives an informal introduction to Android multitasking mechanism between activities and fragments. Section 3 motivates this paper with two open-source apps from F-Droid. Section 4 introduces the AMASS model and formalization of the multitasking mechanism between activities and fragments. Section 5 shows how to extract AMASS models out of Android APK files. Section 6 describes how to encode the semantics of the AMASS models into the reachability problem of finite state machines that can be tackled by the symbolic model checker nuXmv, if the stack heights are assumed to be bounded by a constant. Section 7 is devoted to the validation of the formal semantics of AMASS models against the actual behaviors of Android apps. Section 8 presents the static analysis algorithms for the task/fragment-container unboundedness problems. Section 9 describes the implementation of the static analysis tool, TaskDroid, the benchmarks, and the experiments for the evaluation of TaskDroid. The paper is concluded in Section 10.

A preliminary version of this paper has appeared in the proceedings of APLAS 2019 [HCW+19]. This paper extends the APLAS paper in the following five aspects: (i) Fragments, a crucial feature of Android multitasking which was ignored in [HCW+19], is fully addressed in the current paper, including the formal semantics and static analysis. (ii) The semantics of the multitasking mechanism between activities and fragments for Android 6.0-13.0 are defined in this paper, while only the semantics for Android 7.0 and 8.0 were defined in [HCW+19]. (iii) For the semantics validation, this paper extends the previous work in two ways: the relevant parts of the source code of Android OS are audited; 20 opens-source F-Droid apps are included in addition to the app that is specially designed for the empirical validation. (iv) The models are extracted both statically and dynamically in this paper, while only static model extraction was used in [HCW+19]. (v) The experiments are considerably more thoroughly done with, in particular, more benchmarks.

2 ANDROID MULTITASKING MECHANISM BETWEEN ACTIVITIES AND FRAGMENTS

In this section, we briefly review the core concepts regarding the android multitasking mechanism, i.e., activities and fragments, and the evolution of the back stack.

2.1 Activity, task and task stack

For the purpose of this paper, an Android app can be considered as a collection of *activities*. At any time, there is one activity displayed on the screen of the device. The activities are organized into tasks. A *task* is a collection of activities to carry out a certain job. The running activities within a task are managed as an *activity stack* following the order that an activity is opened. In the activity stack, there are two distinguished activities, i.e., the *root activity* and *top activity*, which are sitting at the bottom and top of the stack respectively.

The Android system normally run multiple tasks which are organized as the *task stack*. The top task is the *foreground* task whiles others are *background* tasks. When a task comes to the foreground, its top activity is displayed on the device screen. When an activity finishes, it is popped from the activity stack. If the activity stack is not empty, the new top activity is displayed on the screen. Otherwise, the task itself finishes in which case it is popped from the task stack. We mention that, the Home screen comes to the foreground when a user presses the Home button (in this case the task stack will be emptied) or when the task stack becomes empty. The task stack is the central data structure for Android multi-tasking mechanism, and we are mostly interested in its evolution in response to activity activation. When an activity is started, there are three basic attributes which determine the resulting task stack: *launch modes, task affinities*, and *intent flags*.

All the activities of an app, as well as their launch modes and task affinities, are defined in the *manifest file* (AndroidManifest.xml). Differently, intent flags are set by caller activities to declare how to activate target activities by calling startActivity() or startActivityForResult() with the intent flags as its arguments. The launch mode attribute specifies one of four modes to launch an activity: standard, singleTop, singleTask, and singleInstance, with standard being the default. A standard or singleTop activity can be instantiated multiple times leading to duplicated activities in a task. In contrast, an activity with the singleTask or singleInstance launch mode should be instantiated only once. Furthermore, an activity with the singleInstance launch mode is always the root activity of a task. While a singleTask activity can contain other standard or singleTop activities in its task, a singleInstance activity does not contain any other activities in its task. It is the only activity in its task; if it starts another activity, that activity is assigned to a different task. The task affinity attribute is of string data type and specifies to which task the activity prefers to belong. By default, all the activities from the same app have the same affinity, the package name of the app (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. Android allows a great degree of flexibility: activities defined in different apps can share a task affinity whilst activities defined in the same app can be assigned with different task affinities.

Android supports inter-component communication via *intents*. An intent is an asynchronous message that activates activities. Android provides two types of intents, *explicit* intents and *implicit* intents. Explicit intents specify directly which activity to activate. Implicit intents, on the other hand, do not directly specify the activities which should be called, it only specifies actions to be performed. For example, an implicit intent with the action "ACTION_VIEW" and the data of the URI "http://www.google.com" will cause a web browser to open a webpage. Android system searches for all activities which are registered for the specific action and the data type. If many activities are found then the user can select which activities to use. In this paper, we restrict our attention to explicit intents since our focus is to understand the evolvement of the back stack when activities are started, and we do not attack the (another challenging) problem of resolving which activity should be started by implicit intents.

Android provides 23 intent flags related to activities (see [int23] for the detailed description of the intent flags). Intent flags are set by caller activities to declare how to activate target activities and are passed to startActivity() or startActivityForResult() as their arguments. In this paper, among the 23 intent flags, we consider the following 10 ones,

- FLAG_ACTIVITY_NEW_TASK,
- FLAG_ACTIVITY_NEW_DOCUMENT,
- FLAG_ACTIVITY_MULTIPLE_TASK,
- FLAG_ACTIVITY_SINGLE_TOP,
- FLAG_ACTIVITY_REORDER_TO_FRONT,
- FLAG_ACTIVITY_CLEAR_TOP,
- FLAG_ACTIVITY_CLEAR_TASK,
- FLAG_ACTIVITY_PREVIOUS_IS_TOP,
- FLAG_ACTIVITY_NO_HISTORY,
- FLAG_ACTIVITY_TASK_ON_HOME.

The other 13 intent flags are ignored, for various reasons as shown in Table 1.

Intent Flags	**Reasons for being ignored in this paper**
FLAG_ACTIVITY_REQUIRE_DEFAULT	Related to implicit intents, which we do not consider in this paper.
FLAG_ACTIVITY_REQUIRE_NON_BROWSER	
FLAG_ACTIVITY_MATCH_EXTERNAL	
FLAG_ACTIVITY_FORWARD_RESULT	Used for transferring results between activities, which we do not consider in this paper.
FLAG_ACTIVITY_RETAIN_IN_RECENTS	Related to the recent app list, which we do not consider it in this paper.
FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS	
FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET	Deprecated from Android 5.0.
FLAG_ACTIVITY_BROUGHT_TO_FRONT	Can only be set by Android operating system, not set by application code.
FLAG_ACTIVITY_LAUNCHED_FROM_HISTORY	
FLAG_ACTIVITY_NO_USER_ACTION	Related to the life-cycle of activities and we do not model the life-cycle of activities in this paper.
FLAG_ACTIVITY_NO_ANIMATION	Used to show some animation when starting an activity and is not related to the evolvement of the back stack.
FLAG_ACTIVITY_LAUNCH_ADJACENT	Related to the split-screen mode, which we do not consider in this paper.
FLAG_ACTIVITY_RESET_TASK_IF_NEEDED	Related to the “allowTaskReparenting” attribute of activities, which we do not consider in this paper.

Table 1. Reasons for ignoring some intent flags in this paper

Besides launch modes, task affinities, and intent flags, there are some other factors that may also affect the behaviors of the task stack. When a caller activity calls startActivity() or startActivityForResult() to start a callee activity, the caller activity can also call the *finish()* procedure so that when the callee activity is started, the caller activity is finished. Moreover, the *real activity*[4] of a task also matters. The real activity of a task is the activity that was pushed into the task as the bottom activity when the task was created. If the real activity of a task is B and the task is not the main task, then attempting to push an instance of B to the task will not modify the task.

[4]The name is inherited from the Android system.

2.2 Fragment, fragment stack, fragment transaction and fragment transaction stack

Importantly, activities are *not* an atomic object and may contain sub-components such as fragments. Previous formalism did not address these, but the current paper will (cf. Section 4). In a nutshell, a fragment represents a modular portion of the user interface within an activity. Related to fragments, *view* is a basic building block of UI in Android. Intuitively, a view is a small rectangular box that responds to user inputs (e.g., EditText, Button, CheckBox, etc.) One can simply understand that a fragment serves as a canvas where different views are aggregated which can, for instance, facilitate reuse. Fragments are stored in a fragment container as a stack, which is called *fragment stacks*. Moreover, an activity may maintain several fragment containers.

At runtime, an app can add, remove, replace fragments in response to user interaction. The add (resp. remove) action will push (resp. pop) a fragment into (resp. out of) a fragment container. The replace action will first remove all the fragments in a fragment container, then push a fragment into it. Multiple fragment actions may be involved in response to one user interaction. Typically, these fragment actions are grouped into *fragment transactions*, where either all the fragment actions are executed, or none of them is executed. The app can choose to push some fragment transactions into a stack called *fragment transaction stack*, which is used to restore the historical states by cancelling the effects of the fragment transactions in the stack, when a user presses the back button later on.

It is worth emphasizing that, unfortunately, the terminologies in literature (such as multitasking, activity stacks, task stacks) are not necessarily consistent. In particular, the notion of multitasking is sometimes used to denote the split-screen multitasking where the screen is split into regions to allow several apps displayed simultaneously, moreover, the notion of back stack is widely used in Android documents, but may (misleadingly) refer to any stack regarding the action of pressing the back button. In this paper, we use multitasking to denote the fundamental mechanism of the Android operating system that utilizes the task stack, a two-tier stack system, to facilitate smooth switching between tasks, even if the device is not in the spit-screen mode. Furthermore, this paper will clarify the notions related to the multitasking mechanism (e.g. tasks and activities) via a proper formalization.

It turns out that there are subtle differences between the multitasking mechanisms of different versions of Android. In this paper, we focus on the multitasking mechanisms of the following versions of Android: 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0 and 13.0.

3 MOTIVATING EXAMPLES

We use two simple apps from F-Droid, namely, "LaunchTime Homescreen"[5] and "ShoppingList"[6], to motivate this paper. We use the two examples to demonstrate that a model of the Android multitasking mechanism where the various factors related to activities and fragments, including launch modes, task affinities, intent flags, the finish() procedure, and fragment actions, are taken into account, enables a more accurate static analysis of Android apps.

3.1 The "LaunchTime Homescreen" app: A motivating example for accurate modeling of factors of activities

The "LaunchTime Homescreen" app contains ten activities. Let us focus on MainActivity and SettingsActivity. The snippet of the source code of the two activities in the "LaunchTime Homescreen" app is shown in Figure 1. The launch modes of MainActivity and SettingsActivity are singleInstance and standard respectively. In line 3942 of the MainActivity.java file (see Figure 1), MainActivity starts SettingsActivity by calling the function startActivity(settingsIntent),

[5] available at https://github.com/quaap/LaunchTime/blob/master/

[6] available at https://github.com/GroundApps/ShoppingList/blob/master/

where the settingsIntent contains the intent flag FLAG_ACTIVITY_NEW_TASK. On the other hand, in line 142 of the SettingsActivity.java file (see Figure 1), SettingsActivity starts MainActivity by calling the function startActivity(main), where all the intent flags are set to be false. Moreover, in line 144 of the same file, finish() is called after MainActivity is started. That is, when MainActivity is started, SettingsActivity is finished.

```
// app/src/main/AndroidManifest.xml
24     <activity
25         android:name=".MainActivity"
26         android:configChanges="orientation|keyboardHidden|screenSize"
27         android:launchMode="singleInstance"
28         android:windowSoftInputMode="stateHidden|adjustPan">
29         <intent-filter>
30             <action android:name="android.intent.action.MAIN" />
               ...
35         </intent-filter>
           ...
47     </activity>
48     <activity
49         android:name=".SettingsActivity"
           ...
55     </activity>

// app/src/main/java/com/quaap/launchtime/MainActivity.java
3937   public static void openSettings(Activity activity) {
3938       Intent settingsIntent = new Intent(activity, SettingsActivity.class);
3939       settingsIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
           ...
3942       activity.startActivity(settingsIntent);
3943   }

// app/src/main/java/com/quaap/launchtime/SettingsActivity.java
139    public boolean onKeyDown(int keyCode, KeyEvent event) {
140        if(keyCode==KeyEvent.KEYCODE_HOME || keyCode==KeyEvent.KEYCODE_MENU) {
141            Intent main = new Intent(this, MainActivity.class);
142            startActivity(main);
143            setResult(RESULT_OK);
144            finish();
145        }
146        return super.onKeyDown(keyCode, event);
147    }
```

Fig. 1. Source code of the F-Droid app “LaunchTime Homescreen”

Let us consider the task unboundedness problem, that is, whether there is a task where the number of activities in the task can be unbounded, that is, arbitrarily large. In the sequel, we show how the AMASS model provides more precise information than the other models so that we can detect the task unboundedness of the “LaunchTime Homescreen” app more accurately.

- If we use the activity transition graph (ATG) to model the “LaunchTime Homescreen” app, then the resulting ATG is just a cycle with two vertices MainActivity and SettingsActivity. Then according to the ATG model, we may conclude that the task can become unbounded, since these two activities can start each other indefinitely.
- If we use the models where the launch modes, task affinities, and intent flags are taken into account, e.g. that in [LHR17], then the launch modes and intent flags are added as the labels of vertices and edges respectively in the ATG model. That is, the vertices are MainActivity(singleInstance) and SettingsActivity(standard), where the vertex labels (launch models) are put in brackets, and the edge from MainActivity to SettingActivity is labeled by start(NTK) (where start and NTK are the abbreviations of startActivity and FLAG_ACTIVITY_NEW_TASK

respectively) and the edge from SettingActivity to MainActivity is labeled by start(⊥) (where ⊥ denotes the fact that all the intent flags are false). In this case, compared to the ATG model, more information is available. For instance, we know that there are at most two tasks, one task holding a unique instance of MainActivity (since its launch mode is singleInstance), and the other task holding the instances of SettingsActivity. Nevertheless, we may still conclude that the task where the instances of SettingsActivity belong to can become unbounded, since SettingsActivity can be started by MainActivity for an unbounded number of times.

- On the other hand, if we use the AMASS model in this paper, then in addition to the launch modes and intent flags, we can also model the finish() procedure. That is, the edge from SettingsActivity to MainActivity is labeled by finishStart(⊥), where finishStart represents the fact that finish() is called after startActivity() is called. In this case, each time when MainActivity is started by SettingsActivity, SettingsActivity is finished simultaneously. As a result, the task holding SettingsActivity contains at most one instance of SettingsActivity. We conclude that the "LaunchTime Homescreen" app does not suffer from the task unboundedness problem actually.

This example motivates us to cover in the AMASS model as much as possible the information about activities, including launch modes, task affinities, and intent flags, as well as the finish() procedure, in order to facilitate a precise static analysis of the Android apps.

3.2 The "ShoppingList" app: A motivating example for accurate modeling of factors of fragments

If a model of Android multitasking mechanism does not capture the fragments, then the fragment-container unboundedness problem will be missed in the static analysis of Android apps. Furthermore, if the fragments are captured, but in an imprecise way, then the static analysis may still be inaccurate. Let us use the "ShoppingList" app to illustrate this point. The "ShoppingList" app comprises two activities, MainActivity and SettingsActivity. The MainActivity contains five fragments. Let us focus on three of them, namely, ErrorFragment, ShoppingListFragment and CacheListFragment. From the snippet of the source code in Figure 2, ErrorFragment can start ShoppingListFragment and CacheListFragment via replace() function (see line 69 and 75 of the ErrorFragment.java file). On the other hand, ShoppingListFragment can start ErrorFragment via replace() function (see line 392 of the ShoppingListFragment file).

- If an imprecise model for fragments, e.g. the AFTG model in [CHGD18], is used, then we may wrongly reports that "ShoppingList" app suffers from the fragment-container unboundedness problem. The AFTG model extends the ATG model by taking the fragments into consideration and consider all the transitions between them. Nevertheless, the AFTG model does not distinguish between the add and replace actions. As a result, from the existence of a cycle between ErrorFragment and ShoppingListFragment in the AFTG model, we may report that the "ShoppingList" app is fragment-container unbounded.
- On the other hand, if the AMASS model is used, then we can add labels to the edges in the AFTG model and distinguish between the add and replace actions. Then we know that the two edges between ErrorFragment and ShoppingListFragment are both labeled by the replace action. As a result, before ErrorFragment or ShoppingList-Fragment is pushed to the fragment container, the fragment container is emptied. We conclude that the cycle between ErrorFragment or ShoppingListFragment does not lead to the fragment-container unboundedness problem.

This example motivates us to capture the information about fragments, in particular, distinguish between different fragment actions, in the definition of the AMASS model.

```
// app/src/main/java/org/janb/shoppinglist/fragments/ErrorFragment.java
63     public void onClick(View view) {
64         android.app.FragmentManager fragmentManager = getFragmentManager();
65         FragmentTransaction transaction = fragmentManager.beginTransaction();
66         switch (view.getId()) {
67             case R.id.error_btn_retry:
68                 ShoppingListFragment listFR = new ShoppingListFragment();
69                 transaction.replace(R.id.fragment_container, listFR);
70                 transaction.addToBackStack(null);
71                 transaction.commit();
72                 break;
73             case R.id.error_btn_cache:
74                 CacheListFragment cacheFR = new CacheListFragment();
75                 transaction.replace(R.id.fragment_container, cacheFR, "CACHE_FRAGMENT");
76                 transaction.addToBackStack(null);
77                 transaction.commit();
78                 break;
               ...
83         }
84     }
// app/src/main/java/org/janb/shoppinglist/fragments/ShoppingListFragment.java
378    public void onError(ResponseHelper error) {
           ...
385        ErrorFragment errFR;
           ...
390        FragmentManager fragmentManager = getFragmentManager();
391        FragmentTransaction transaction = fragmentManager.beginTransaction();
392        transaction.replace(R.id.fragment_container, errFR);
393        transaction.addToBackStack(null);
394        transaction.commitAllowingStateLoss();
395    }
```

Fig. 2. Source code of the F-Droid app "ShoppingList"

4 ANDROID MULTITASKING STACK SYSTEMS

In this section, we introduce **A**ndroid **M**ultit**A**sking **S**tack **S**ystem (AMASS), a formal model to capture the Android multitasking mechanism. Our presentation is inspired by the previous work [CHS+18, HCW+19], but the model significantly deviates from the ASM therein. Throughout the paper, we let $[m] = \{1, \cdots, m\}$, $\mathbb{N}$ be the set of natural numbers, and $\mathbb{N}_{>0}$ be the set of positive natural numbers.

Following the overview of Section 2, we shall concentrate on the launch mode, the task affinity, the intent flags when an activity is launched, and the fragment transaction when a fragment is started.

- There are four launch modes in Android: "standard (STD)", "singleTop (STP)", "singleTask (STK)" and "singleInstance (SIT)". We shall consider all of them.
- As mentioned before, we focus on 10 intent flags in this paper, namely,
  - FLAG_ACTIVITY_NEW_TASK (NTK),
  - FLAG_ACTIVITY_NEW_DOCUMENT (NDM),
  - FLAG_ACTIVITY_MULTIPLE_TASK (MTK),
  - FLAG_ACTIVITY_SINGLE_TOP (STP),
  - FLAG_ACTIVITY_REORDER_TO_FRONT (RTF),
  - FLAG_ACTIVITY_CLEAR_TOP (CTP),
  - FLAG_ACTIVITY_CLEAR_TASK (CTK),
  - FLAG_ACTIVITY_PREVIOUS_IS_TOP (PIT),
  - FLAG_ACTIVITY_NO_HISTORY (NOH),

    - FLAG_ACTIVITY_TASK_ON_HOME (TOH).
- There are three actions in fragment transaction: "add (ADD)", "replace (REP)" and "remove (REM)", and we consider all of them. The semantics of these actions are related to the identifiers of fragment instances which we assume are from $\mathbb{N}_{>0}$.

Moreover, for fragment transactions, we use the tag TS (resp. NTS) to represent the fact that a fragment transaction will be pushed (resp. will not be pushed) to the fragment transaction stack.

All the abbreviations and their full names are summarized in Table 2 to facilitate the later references. Note that STP is used as abbreviations both for the SingleTop launch mode and for the FLAG_ACTIVITY_SINGLE_TOP intent flag. This is acceptable since it is usually clear from the context whether STP denotes a launch mode or an intent flag.

Abbreviation	**Full name**
STD	Standard
STP	SingleTop
STK	SingleTask
SIT	SingleInstance
STP	FLAG_ACTIVITY_SINGLE_TOP
CTP	FLAG_ACTIVITY_CLEAR_TOP
RTF	FLAG_ACTIVITY_REORDER_TO_FRONT
NTK	FLAG_ACTIVITY_NEW_TASK
NDM	FLAG_ACTIVITY_NEW_DOCUMENT
MTK	FLAG_ACTIVITY_MULTIPLE_TASK
CTK	FLAG_ACTIVITY_CLEAR_TASK
PIT	FLAG_ACTIVITY_PREVIOUS_IS_TOP
NOH	FLAG_ACTIVITY_NO_HISTORY
TOH	FLAG_ACTIVITY_TASK_ON_HOME
ADD	add
REP	replace
REM	remove
TS	added to the fragment transaction stack
NTS	not added to the fragment transaction stack

Table 2. Abbreviations and their full names

Let $\mathcal{F} = \{\mathsf{NTK}, \mathsf{NDM}, \mathsf{MTK}, \mathsf{STP}, \mathsf{RTF}, \mathsf{CTP}, \mathsf{CTK}, \mathsf{PIT}, \mathsf{NOH}, \mathsf{TOH}\}$ denote the set of intent flags, $\mathscr{B}(\mathcal{F})$ denote the set of formulae $\phi = \bigwedge_{F \in \mathcal{F}} \theta_F$, where $\theta_F = F$ or $\neg F$. For convenience, we use $\perp$ to denote $\bigwedge_{F \in \mathcal{F}} \neg F$.

Definition 1 (Android Multitasking Stack System, AMASS). *An AMASS is a tuple*

$$\mathcal{M} = (\mathsf{Act}, A_0, \mathsf{Frg}, \mathsf{Lmd}, \mathsf{Aft}, \mathsf{Ctn}, \Delta),$$

where

- Act *is a finite set of activities, and* $A_0 \in \mathsf{Act}$ *is the main activity, let* $m = |\mathsf{Act}|$*,*
- Frg *is a finite set of fragments,*
- $\mathsf{Lmd} : \mathsf{Act} \to \{\mathsf{STD}, \mathsf{STP}, \mathsf{STK}, \mathsf{SIT}\}$ *is the launch-mode function,*
- $\mathsf{Aft} : \mathsf{Act} \to [m]$ *is the task-affinity function,*

- $\mathsf{Ctn} : \mathsf{Act} \to \mathbb{N}^*$ *is the fragment container function that assigns to each activity a finite sequence of mutually-distinct fragment container identifiers,*
- $\Delta \subseteq (\mathsf{Act} \cup \mathsf{Frg}) \times (\mathsf{ActInst} \cup \mathsf{FrgInst}) \cup \{\mathsf{back}\}$ *is a finite set of transition rules. Here* $\mathsf{ActInst} =$

$$\{\alpha(A, \phi) \mid \alpha \in \{\mathsf{start}, \mathsf{finishStart}\}, A \in \mathsf{Act}, \phi \in \mathscr{B}(\mathcal{F})\}$$

and $\mathsf{FrgInst} = \{\mu[T] \mid \mu \in \{\mathsf{TS}, \mathsf{NTS}\}, T \in \mathcal{T}\}$ *where* $\mathcal{T}$ *is a finite set of fragment transactions of the form* $(\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))$ *where for every* $j \in [k]$, $\beta_j \in \{\mathsf{ADD}, \mathsf{REP}, \mathsf{REM}\}$, $F_j \in \mathsf{Frg}$, $i_j \in \mathbb{N}$, *and* x_j *is a variable storing the identifiers of fragment instances.*

Moreover, we consider several sub-models of AMASS, namely, activity-oriented *AMASS (AMASS$_{ACT}$) and* fragment-oriented *AMASS (AMASS$_{FRG}$), where all the transition rules are on the activity-level and fragment-level respectively. More precisely, an AMASS$_{ACT}$ (resp. AMASS$_{FRG}$) is an AMASS where all transitions are* activity-oriented, *that is,* $\Delta \subseteq \mathsf{Act} \times \mathsf{ActInst} \cup \{\mathsf{back}\}$ *(resp.* fragment-oriented, $\Delta \subseteq \mathsf{Frg} \times \mathsf{FrgInst} \cup \{\mathsf{back}\}$*).*

For readability, we write a transition rule $(A, \alpha(B, \phi)) \in \Delta$ as $A \xrightarrow{\alpha(\phi)} B$ and $(A, \mu[T]) \in \Delta$ as $A \xrightarrow{\mu} T$. Therefore, all the transitions in an AMASS$_{ACT}$ (resp. AMASS$_{FRG}$), except back, are of the form $A \xrightarrow{\alpha(\phi)} B$ (resp. $A \xrightarrow{\mu} T$).

The rest of this section is devoted to the semantics of AMASS. We shall first define the semantics of AMASS for Android 13.0, and consider the semantics for other versions later.

4.1 Semantics of AMASS for Android 13.0

As the model of AMASS involves both activities and fragments, its semantics is rather involved. In order to ease the understanding, we separate the concerns and define the formal semantics of AMASS$_{ACT}$ and AMASS$_{FRG}$ respectively. (The long and detailed definition of the formal semantics of AMASS is presented in the appendix.)

4.1.1 Semantics of AMASS$_{ACT}$. It turns out that the semantics of AMASS$_{ACT}$ is still complicated due to the complex interplay between launch modes and intent flags. Therefore, in the sequel, we separate the concerns further and consider the two sub-models of AMASS$_{ACT}$, namely AMASS$_{ACT,LM}$ and AMASS$_{ACT,IF}$, which focus on launch modes and intent flags of AMASS$_{ACT}$ respectively. More precisely,

- an AMASS$_{ACT,LM}$ is an AMASS$_{ACT}$ where all the transition rules $A \xrightarrow{\alpha(\phi)} B$ (except back) satisfy that $\phi = \perp$,
- an AMASS$_{ACT,IF}$ is an AMASS$_{ACT}$ where all the transition rules $A \xrightarrow{\alpha(\phi)} B$ (except back) satisfy that $\mathsf{Lmd}(A) = \mathsf{STD}$.

To ease the understanding, in the main text, we shall only define the formal semantics of the two sub-models AMASS$_{ACT,LM}$ and AMASS$_{ACT,IF}$, and omit the definition of the formal semantics of AMASS$_{ACT}$, since the definition of the semantics of AMASS$_{ACT}$ is rather tedious and we think that AMASS$_{ACT,LM}$ and AMASS$_{ACT,IF}$ are already sufficient to understand the meanings of the launch modes and intent flags.

To simplify the presentation, in $A \xrightarrow{\alpha(\phi)} B$, we assume that α is start. The definition of the semantics for the case that α is finishStart can be found in the appendix.

Semantics of AMASS$_{ACT,LM}$.

We start with the semantics of AMASS$_{ACT,LM}$ and assume that $\mathcal{M}$ is an AMASS$_{ACT,LM}$. We first introduce some notations.

Tasks and configurations. A *task* of $\mathcal{M}$ is represented by its activity stack and is encoded as a word $S = [A_1, \cdots, A_n] \in \mathsf{Act}^+$, with A_1 (resp. A_n) as the top (resp. bottom) activity of S, n is called the *height* of S.

A *configuration* of $\mathcal{M}$ is encoded as a sequence $\rho = (\Omega_1, \cdots, \Omega_n)$, and for each $i \in [n]$, $\Omega_i = (S_i, A_i, \zeta_i)$, $S_i \in \mathsf{Act}^*$ is a task, $A_i \in \mathsf{Act}$ is the real activity of S_i, $\zeta_i \in \{\mathsf{MAIN}, \mathsf{STK}, \mathsf{SIT}\}$ represents how the task S_i is launched. For any activity A, we refer to an A-task as a task whose real activity is A. The tasks S_1 and S_n are called the top and the bottom task respectively. (Intuitively, S_1 is the foreground task.) The symbol ε is used to denote the empty task stack. The *affinity of a task* is defined as the affinity of its real activity. A task (S_i, A_i, ζ_i) in ρ is called an SIT-task if $\zeta_i = \mathsf{SIT}$.

A task is called the *main task* of the task stack if it is the first task that was created when launching the app. Note that the current task stack may *not* contain the main task, since it may have been popped out from the task stack. This notion is introduced since the semantics of AMASS is also dependent on whether the task stack contains the main task.

Let $\mathrm{Conf}_{\mathcal{M}}$ denote the set of configurations of $\mathcal{M}$. The *initial* configuration of $\mathcal{M}$ is $(([A_0], A_0, \mathsf{MAIN}))$. The *height* of a configuration ρ is defined as $\max_{i \in [m]} |S_i|$, where $|S_i|$ is the height of S_i. By convention, the height of ε is defined as 0.

Before presenting the formal semantics of $\mathsf{AMASS}_{ACT,LM}$, we present its intuitions.

Intuitions of the launch modes. We call an activity of the launch mode STD as an STD-activity, similarly for STP, STK and SIT.

- The STD mode: When a new STD-activity is started, it will be pushed into the top task.
- The STP mode: When a new activity of the STP mode is started, if the activity is already at the top of the top task, it will reuse this activity. Otherwise, a new activity will be pushed into the top task.
- The STK mode: When a new activity of the STK mode is started, it will create the activity at the root of a new task or locates the activity on an existing task with the same affinity. If the activity already exists, then all the activities above it are removed from the task. Otherwise, a new activity will be pushed into the task.
- The SIT mode: Similar to STK, but if such an activity already exists, it will reuse this activity, moreover there is only one activity in the task which was created by starting the same activity.

Task allocation mechanism. The intuitions of the launch modes are actually not that precise. For instance, when a new STD-activity is stated by an SIT-activity, the STD-activity will not necessarily be pushed to the top task. Task allocation mechanism is to specify to which task will it be allocated when an activity is launched. Via extensive experiments, we identify a crucial notion, i.e., real activity of tasks, which plays a pivotal role in such a mechanism.

Generally speaking, for an activity B which is not to land on the top task, the following three steps will apply: (1) If there is any task whose real activity is B, then B will be put on the task. (2) Otherwise, if there is any task whose real activity has the same *task affinity* as B, then B will be put on the task. (3) Otherwise, a new task is created to hold B. In the first two cases, if there are multiple instances, the first occurrence starting from the top task will be selected.

Real activity and main task. When the caller activity is an SIT activity and the callee activity is an STD or STP activity B, B will *not* always be pushed into the task S_i which is specified according to the *task allocation mechanism*. Generally speaking, the following steps will apply: (1) If the real activity of S_i is *not* B, then B will be pushed into S_i. (2) Otherwise, if S_i is the main task, that is, $\zeta_i = \mathsf{MAIN}$, then B will be pushed into S_i. (If S_i is not the main task, then B will not be pushed.)

Then we introduce some auxiliary functions and predicates to be used in the formal semantics of $\mathsf{AMASS}_{ACT,LM}$.

Auxiliary functions and predicates. To specify the transition relation precisely and concisely, we define the following functions and predicates. Let $\rho = (\Omega_1, \cdots, \Omega_n)$ be a configuration with $\Omega_i = (S_i, A_i, \zeta_i)$ for each $i \in [n]$, and $S = [B_1, \cdots, B_m]$ be a task.

- $\mathsf{TopAct}(S) = B_1$, $\mathsf{BtmAct}(S) = B_m$,
- $\mathsf{TopTsk}(\rho) = S_1$, $\mathsf{TopAct}(\rho) = \mathsf{TopAct}(\mathsf{TopTsk}(\rho))$,
- $\mathsf{Push}(\rho, B) = (([B] \cdot S_1, A_1, \zeta_1), \Omega_2, \cdots, \Omega_n)$,
- $\mathsf{ClrTop}(\rho, B) = (([B] \cdot S_1'', A_1, \zeta_1), \Omega_2, \cdots, \Omega_n)$ if $S_1 = S_1' \cdot S_1''$ with $S_1' \in (\mathsf{Act} \setminus \{B\})^* B$,
- $\mathsf{ClrTsk}(\rho, B) = (([B], A_1, \zeta_1), \Omega_2, \cdots, \Omega_n)$,
- $\mathsf{MvTsk2Top}(\rho, i) = (\Omega_i, \Omega_1, \cdots, \Omega_{i-1}, \Omega_{i+1}, \cdots, \Omega_n)$,
- $\mathsf{NewTsk}(\rho, B, \zeta) = (([B], B, \zeta), \Omega_1, \cdots, \Omega_n)$,
- $\mathsf{GetRealTsk}(\rho, B) = S_i$ such that $i \in [n]$ is the *minimum* index satisfying $A_i = B$ if such an index i exists; $\mathsf{GetRealTsk}(\rho, B) = *$ otherwise,
- $\mathsf{GetTsk}(\rho, B) = S_i$ such that $i \in [n]$ is the *minimum* index satisfying $\mathsf{Aft}(A_i) = \mathsf{Aft}(B)$ and $\zeta_i \in \{\mathsf{MAIN}, \mathsf{STK}\}$, if such an index i exists; $\mathsf{GetTsk}(\rho, B) = *$ otherwise.

The formal semantics of $\mathcal{M}$ will be defined as a transition relation $\xrightarrow{\mathcal{M}}$. We first use the following example to illustrate the semantics.

Example 4.1. Let $\mathcal{M} = (\mathsf{Act}, A, \mathsf{Frg}, \mathsf{Lmd}, \mathsf{Aft}, \mathsf{Ctn}, \Delta)$ be an $\mathsf{AMASS}_{ACT,LM}$, where $\mathsf{Act} = \{A, B, C, D\}$, the functions Lmd and Aft are defined in Table 3.

Activity	Lmd	Aft
A	STK	1
B	STP	2
C	SIT	1
D	STD	2

Table 3. Attributes of activities

Moreover, $\Delta = \{\mathsf{back}, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$, where $\tau_1 = A \xrightarrow{\mathsf{start}(\bot)} B$, $\tau_2 = B \xrightarrow{\mathsf{start}(\bot)} C$, $\tau_3 = C \xrightarrow{\mathsf{start}(\bot)} D$, $\tau_4 = D \xrightarrow{\mathsf{start}(\bot)} A$, $\tau_5 = B \xrightarrow{\mathsf{start}(\bot)} B$. Then the configurations reachable from the initial configuration $(([A], A, \mathsf{MAIN}))$ by executing the transition rules from Δ are illustrated in Figure 3, where the vertices denote the configurations and the edges denote the elements of $\xrightarrow{\mathcal{M}}$. For instance,

- if the transition rule $A \xrightarrow{\mathsf{start}(\bot)} B$ is applied to the configuration $(([A], A, \mathsf{MAIN}))$, then B is pushed, since $\mathsf{Lmd}(B) = \mathsf{STP}$ and $A \neq B$, resulting in the configuration $(([BA], A, \mathsf{MAIN}))$,
- if $B \xrightarrow{\mathsf{start}(\bot)} B$ is applied to the configuration $(([BA], A, \mathsf{MAIN}))$, then B will not be pushed, since $\mathsf{Lmd}(B) = \mathsf{STP}$ and the top activity of the top task is B,
- if $B \xrightarrow{\mathsf{start}(\bot)} C$ is applied to the configuration $(([BA], A, \mathsf{MAIN}))$, then a new task $([C], C, \mathsf{SIT})$ is created, since $\mathsf{Lmd}(C) = \mathsf{SIT}$, resulting in the configuration $(([C], C, \mathsf{SIT}), ([BA], A, \mathsf{MAIN}))$,
- if $C \xrightarrow{\mathsf{start}(\bot)} D$ is applied to the configuration $(([C], C, \mathsf{SIT}), ([BA], A, \mathsf{MAIN}))$, then a new task $([D], D, \mathsf{STK})$ is created, since $\mathsf{Lmd}(C) = \mathsf{SIT}$, $\mathsf{Lmd}(D) = \mathsf{STD}$ and $\mathsf{Aft}(D) = 2 \neq \mathsf{Aft}(A)$, resulting in the configuration $(([D], D, \mathsf{STK}), ([C], C, \mathsf{SIT}), ([BA], A, \mathsf{MAIN}))$,

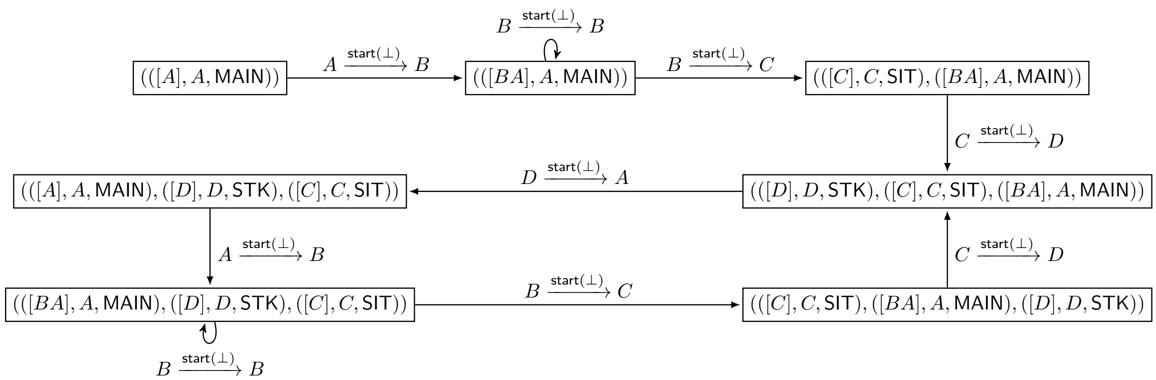

Fig. 3. Configurations reachable from the initial configuration $(([A], A, \mathsf{MAIN}))$ in a $\mathsf{AMASS}_{ACT,LM}$ $\mathcal{M}$

- if $D \xrightarrow{\mathsf{start}(\perp)} A$ is applied to the configuration $(([D], D, \mathsf{STK}), ([C], C, \mathsf{SIT}), ([BA], A, \mathsf{MAIN}))$, then the task $([BA], A, \mathsf{MAIN})$ is moved to the top and all the activities above A, which is B here, are removed from the task, since $\mathsf{Lmd}(A) = \mathsf{STK}$,

$$\mathsf{GetRealTsk}((([D], D, \mathsf{STK}), ([C], C, \mathsf{SIT}), ([BA], A, \mathsf{MAIN})), A) = ([BA], A, \mathsf{MAIN}),$$

  and A occurs in $([BA], A, \mathsf{MAIN})$, resulting in the configuration $(([A], A, \mathsf{MAIN}), ([D], D, \mathsf{STK}), ([C], C, \mathsf{SIT}))$,
- ...
- if $C \xrightarrow{\mathsf{start}(\perp)} D$ is applied to the configuration $(([C], C, \mathsf{SIT}), ([BA], A, \mathsf{MAIN}), ([D], D, \mathsf{STK}))$, then the task $([D], D, \mathsf{STK})$ is moved to the top, but D will not be pushed, since $\mathsf{Lmd}(C) = \mathsf{SIT}$, $\mathsf{Lmd}(D) = \mathsf{STD}$,

$$\mathsf{GetRealTsk}((([C], C, \mathsf{SIT}), ([BA], A, \mathsf{MAIN}), ([D], D, \mathsf{STK})), D) = ([D], D, \mathsf{STK}),$$

  and $([D], D, \mathsf{STK})$ is not the main task, resulting in the configuration $(([D], D, \mathsf{STK}), ([C], C, \mathsf{SIT}), ([BA], A, \mathsf{MAIN}))$.

Note that for $\mathcal{M}$, there are only finitely many configurations reachable from the initial configuration, which may not be the case for $\mathsf{AMASS}_{ACT,LM}$ in general.

From the informal description and the example above, we have already gotten an intuitive understanding of the semantics of $\mathcal{M}$. To facilitate a precise understanding of the semantics of $\mathsf{AMASS}_{ACT,LM}$, let us formally define the semantics of $\mathcal{M}$ as a transition relation $\xrightarrow{\mathcal{M}}$ in the sequel.

Transition relation. We define the relation $\xrightarrow{\mathcal{M}}$ which comprises the quadruples $(\rho, \tau, \rho') \in \mathsf{Conf}_{\mathcal{M}} \times \Delta \times \mathsf{Conf}_{\mathcal{M}}$ to formalise the semantics of $\mathcal{M}$. For readability, we write $(\rho, \tau, \rho') \in \xrightarrow{\mathcal{M}}$ as $\rho \xrightarrow[\tau]{\mathcal{M}} \rho'$.

Let $\rho = ((S_1, A_1, \zeta_1), \cdots, (S_n, A_n, \zeta_n))$ be the current configuration for some $n \geq 1$ and $\mathsf{TopAct}(\rho) = A$. Moreover, let $S_1 = [A'_1, \cdots, A'_m]$. Evidently, $A = A'_1$.

If $\tau = \mathsf{back}$, then $\rho' = ((S'_1, A_1, \zeta_1), (S_2, A_2, \zeta_2) \cdots, (S_n, A_n, \zeta_n))$ if $m > 1$, where $S'_1 = [A'_2, \cdots, A'_m]$, and $\rho' = ((S_2, A_2, \zeta_2) \cdots, (S_n, A_n, \zeta_n))$ otherwise.

Then let us consider $\tau = A \xrightarrow{\mathsf{start}(\phi)} B$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{STD}}$

- If $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, then $\rho' = \mathsf{Push}(\rho, B)$.
- If $\mathsf{Lmd}(A) = \mathsf{SIT}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq \mathsf{MAIN}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i = \mathsf{MAIN}$, or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
  - if $\mathsf{GetTsk}(\rho, B) = *$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{STK})$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{STP}}$

- If $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, then
  - if $A = B$, then $\rho' = \rho$,
  - otherwise, $\rho' = \mathsf{Push}(\rho, B)$.
- If $\mathsf{Lmd}(A) = \mathsf{SIT}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq \mathsf{MAIN}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i = \mathsf{MAIN}$, or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$,
    - if $\mathsf{TopAct}(S_i) = B$, then $\rho' = \mathsf{MvAct2Top}(\rho, i)$,
    - otherwise $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
  - if $\mathsf{GetTsk}(\rho, B) = *$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{STK})$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{SIT}}$

- If $\mathsf{GetRealTsk}(\rho, B) = S_i$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$.
- If $\mathsf{GetRealTsk}(\rho, B) = *$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{SIT})$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{STK}}$

- If $\mathsf{GetRealTsk}(\rho, B) = S_i$, or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$ then
  - if $B \notin S_i$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
  - if $B \in S_i$, then $\rho' = \mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B)$.
- If $\mathsf{GetTsk}(\rho, B) = *$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{STK})$.

From the definition of the semantics, we can see that a new task $([B], B, \mathsf{STK})$ is created, not only when $\mathsf{Lmd}(B) = \mathsf{STK}$, but also when $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\mathsf{Lmd}(B) = \mathsf{STD}$.

Semantics of AMASS$_{ACT,IF}$.

The intuitions of the intent flags are given in Table 4. We would like to warn that although the intuitions of these intent flags may help the readers to get some preliminary idea of their meanings, before diving directly into the formal semantics, they are nonetheless inaccurate, especially when different flags may interfere with each other.

To ease the presentation, let us assume that $\phi \models \neg\mathsf{NOH}$ in $A \xrightarrow{\mathsf{start}(\phi)} B$. The full semantics of AMASS$_{ACT,IF}$ where the situation $\phi \models \mathsf{NOH}$ is taken into consideration can be found in Appendix B.

Let $\mathcal{M}$ be an AMASS$_{ACT,IF}$, namely, $\mathsf{Lmd}(A) = \mathsf{STD}$ for each activity A.

To define the semantics of $\mathcal{M}$, the concept of configurations is adapted from the definition of configurations for AMASS$_{ACT,LM}$ as follows: A configuration of $\mathcal{M}$ is still encoded as a sequence $\rho = (\Omega_1, \cdots, \Omega_n)$ such that for each $i \in [n]$, $\Omega_i = (S_i, A_i, \zeta_i)$, where $S_i \in \mathsf{Act}^*$ is a task, $A_i \in \mathsf{Act}$ is the real activity of S_i, but $\zeta_i \in \{\mathsf{MAIN}, \mathsf{NTK}, \mathsf{NDM}\}$. Intuitively, NTK in ζ_i plays the same role as STK in AMASS$_{ACT,LM}$, and NDM is added for the intent flag NDM, moreover, SIT disappears since in AMASS$_{ACT,IF}$, the launch modes of all activities are assumed to be STD.

Then we introduce some additional auxiliary functions and predicates to be used in the formal semantics of $\mathcal{M}$.

Intent Flags	Abbreviation	Intuition
FLAG_ACTIVITY_SINGLE_TOP	STP	If it is set, it has the same effect as starting an activity of the STP launch mode.
FLAG_ACTIVITY_CLEAR_TOP	CTP	If it is set, all the activities above the topmost occurrence of the started activity in the top task will be removed.
FLAG_ACTIVITY_REORDER_TO_FRONT	RTF	If it is set, it will check for the existence of the started activity in the task. If an instance of the activity exists, then the topmost occurrence of this activity will be moved to the top of this task. Otherwise, a new instance of the activity will be pushed into the top task.
FLAG_ACTIVITY_NEW_TASK	NTK	If it is set, it will look for an existing task to put the started activity according to the task allocation mechanism. If such a task exists, the task will be moved to the top and a new instance of the activity is pushed to the task, otherwise a new task is created to put this activity.
FLAG_ACTIVITY_NEW_DOCUMENT	NDM	If it is set, its behavior is similar to the STK launch mode, but the task allocation mechanism is slightly different, namely, it will look for an existing task by only using real activities of tasks, but not affinities.
FLAG_ACTIVITY_MULTIPLE_TASK	MTK	It is usually used together with NTK or NDM. If it is set, then it will always create a new task to put the started activity, no matter whether there already exists a task of the same affinity as this activity.
FLAG_ACTIVITY_CLEAR_TASK	CTK	It is usually used together with NTK or NDM. If it is set, it will remove all the activities in the task and push the started activity to the task.
FLAG_ACTIVITY_NO_HISTORY	NOH	If it is set, when the started activity becomes a non-topmost activity in the future, the activity will be removed.
FLAG_ACTIVITY_PREVIOUS_IS_TOP	PIT	If it is set, then when starting this activity, the activity immediately below the topmost activity will play the role of the topmost activity.
FLAG_ACTIVITY_TASK_ON_HOME	TOH	It is usually used together with NTK or NDM. If it is set, then only the top task, which is either a newly created task or an existing task moved to the top, is kept, and all the other tasks are removed.

Table 4. Intuitions of intent flags

Auxiliary functions and predicates. Let $\rho = (\Omega_1, \cdots, \Omega_n)$ be a configuration with $\Omega_i = (S_i, A_i, \zeta_i)$ for each $i \in [n]$, and $S = [B_1, \cdots, B_m]$ be a task. The following additional auxiliary functions are defined.

- $\mathsf{PreAct}(S) = B_2$ if $m > 1$, $\mathsf{PreAct}(S) = B_1$ otherwise,
- $\mathsf{PreAct}(\rho) = \mathsf{PreAct}(\mathsf{TopTsk}(\rho))$,
- $\mathsf{MvAct2Top}(\rho, B) = (([B] \cdot S_1' \cdot S_1'', A_1, \zeta_1), \Omega_2, \cdots, \Omega_n)$, if $S_1 = S_1' \cdot [B] \cdot S_1''$ with $S_1' \in (\mathsf{Act} \setminus \{B\})^*$,

Intuitively, $\mathsf{PreAct}(S)$ and $\mathsf{PreAct}(\rho)$ are used for defining the semantics of PIT, and $\mathsf{MvAct2Top}(\rho, B)$ is used for defining the semantics of RTF. Moreover, the function GetTsk is adapted by replacing STK with NTK: $\mathsf{GetTsk}(\rho, B) = S_i$

such that $i \in [n]$ is the *minimum* index satisfying $\mathsf{Aft}(A_i) = \mathsf{Aft}(B)$ and $\zeta_i \in \{\mathsf{MAIN}, \mathsf{NTK}\}$, if such an index i exists; $\mathsf{GetTsk}(\rho, B) = *$ otherwise.

Before the formal definition, let us use the following example to illustrate the semantics of the $\mathsf{AMASS}_{ACT,IF}$.

Example 4.2. Let $\mathcal{M} = (\mathsf{Act}, A, \mathsf{Frg}, \mathsf{Lmd}, \mathsf{Aft}, \mathsf{Ctn}, \Delta)$ be an $\mathsf{AMASS}_{ACT,IF}$, where $\mathsf{Act} = \{A, B, C, D, E, F, G\}$, and for each $A' \in \mathsf{Act}$, $\mathsf{Lmd}(A') = \mathsf{STD}$ and $\mathsf{Aft}(A') = 1$. Moreover, $\Delta = \{\mathsf{back}\} \cup \{\tau_i \mid 1 \le i \le 9\}$, where $\tau_1 = A \xrightarrow{\mathsf{start}(\mathsf{CTP})} B$, $\tau_2 = B \xrightarrow{\mathsf{start}(\perp)} C$, $\tau_3 = B \xrightarrow{\mathsf{start}(\mathsf{NTK} \wedge \mathsf{MTK})} D$, $\tau_4 = B \xrightarrow{\mathsf{start}(\mathsf{NDM})} F$, $\tau_5 = C \xrightarrow{\mathsf{start}(\mathsf{NTK})} A$, $\tau_6 = D \xrightarrow{\mathsf{start}(\mathsf{STP})} D$, $\tau_7 = D \xrightarrow{\mathsf{start}(\mathsf{RTF})} E$, $\tau_8 = E \xrightarrow{\mathsf{start}(\mathsf{STP} \wedge \mathsf{PIT})} D$, $\tau_9 = E \xrightarrow{\mathsf{start}(\mathsf{NTK} \wedge \mathsf{CTK})} F$.

The $\xrightarrow{\mathcal{M}}$ relation on the set of configurations that are reachable from the initial configuration $(([A], A, \mathsf{MAIN}))$ is illustrated in Figure 4.

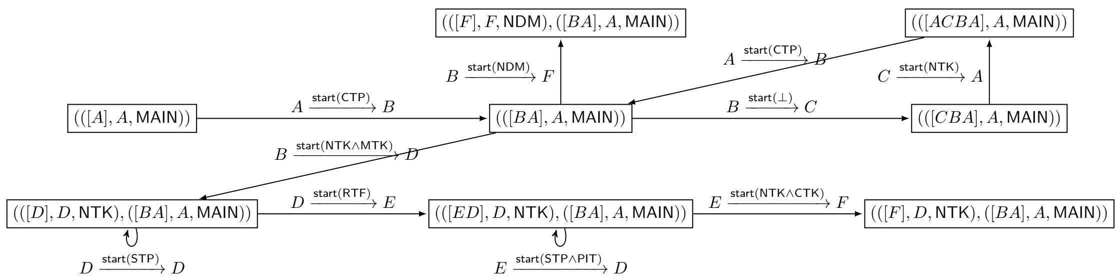

Fig. 4. Configurations that are reachable from the initial configuration $(([A], A, \mathsf{MAIN}))$ in $\mathcal{M}$

For instance,

- from the configuration $(([A], A, \mathsf{MAIN}))$, after executing the transition rule $A \xrightarrow{\mathsf{start}(\mathsf{CTP})} B$, because $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM}$, B does not occur in the top task $[A]$, we have $\rho' = \mathsf{Push}((([A], A, \mathsf{MAIN})), B) = (([BA], A, \mathsf{MAIN}))$,
- from the configuration $(([ACBA], A, \mathsf{MAIN}))$, after executing $A \xrightarrow{\mathsf{start}(\mathsf{CTP})} B$, because $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM}$, B occurs in the top task $[ACBA]$, we have $\rho' = \mathsf{ClrTop}((([ACBA], A, \mathsf{MAIN})), B) = (([BA], A, \mathsf{MAIN}))$,
- from the configuration $(([BA], A, \mathsf{MAIN}))$, after executing $B \xrightarrow{\mathsf{start}(\perp)} C$, because $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{CTP} \wedge \neg\mathsf{RTF} \wedge \neg\mathsf{STP}$,and we have $\rho' = \mathsf{Push}((([BA], A, \mathsf{MAIN})), C) = (([CBA], A, \mathsf{MAIN}))$,
- from the configuration $(([BA], A, \mathsf{MAIN}))$, after executing $B \xrightarrow{\mathsf{start}(\mathsf{NDM})} F$, because $\phi \models \mathsf{NDM} \wedge \neg\mathsf{MTK}$, $\mathsf{GetRealTsk}(\rho, F) = *$, we have $\rho' = \mathsf{NewTsk}((([BA], A, \mathsf{MAIN})), F, \mathsf{NDM}) = (([F], F, \mathsf{NDM}), ([BA], A, \mathsf{MAIN}))$,
- from the configuration $(([BA], A, \mathsf{MAIN}))$, after executing $B \xrightarrow{\mathsf{start}(\mathsf{NTK} \wedge \mathsf{MTK})} D$, because $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \mathsf{MTK}$, we have

$$\rho' = \mathsf{NewTsk}((([BA], A, \mathsf{MAIN})), D, \mathsf{NTK}) = (([D], D, \mathsf{NTK}), ([BA], A, \mathsf{MAIN})),$$

- from the configuration $(([CBA], A, \mathsf{MAIN}))$, after executing $C \xrightarrow{\mathsf{start}(\mathsf{NTK})} A$, because $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{MTK} \wedge \neg\mathsf{CTP} \wedge \neg\mathsf{RTF} \wedge \neg\mathsf{STP}$, $\mathsf{GetRealTsk}(\rho, A) = S_1$, $\zeta_1 = \mathsf{MAIN}$, we have

$$\rho' = \mathsf{Push}((([CBA], A, \mathsf{MAIN})), A) = (([ACBA], A, \mathsf{MAIN})),$$

- from the configuration $(([D], D, \mathsf{NTK}), ([BA], A, \mathsf{MAIN}))$, after executing $D \xrightarrow{\mathsf{start}(\mathsf{STP})} D$, because $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{CTP} \wedge \neg\mathsf{RTF} \wedge \mathsf{STP}$ and D is the top activity of the top task $[D]$, we have $\rho' = \rho$,

- from the configuration $(([D], D, \mathsf{NTK}), ([BA], A, \mathsf{MAIN}))$, after executing $D \xrightarrow{\mathsf{start}(\mathsf{RTF})} E$, because $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{CTP} \wedge \mathsf{RTF}$ and E dose not occur in the top task $[D]$, we have

$$\rho' = \mathsf{Push}((([D], D, \mathsf{NTK}), ([BA], A, \mathsf{MAIN})), E) = (([ED], D, \mathsf{NTK}), ([BA], A, \mathsf{MAIN})),$$

- from the configuration $(([ED], D, \mathsf{NTK}), ([BA], A, \mathsf{MAIN}))$, after executing $E \xrightarrow{\mathsf{start}(\mathsf{STP}\wedge\mathsf{PIT})} D$, because $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{CTP} \wedge \neg\mathsf{RTF} \wedge \mathsf{STP} \wedge \mathsf{PIT}$ and $D = \mathsf{PreAct}([ED])$, we have $\rho' = \rho$,
- from the configuration $(([ED], D, \mathsf{NTK}), ([BA], A, \mathsf{MAIN}))$, after executing $E \xrightarrow{\mathsf{start}(\mathsf{NTK}\wedge\mathsf{CTK})} F$, because $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{MTK} \wedge \mathsf{CTK}$, $\mathsf{GetRealTsk}(\rho, F) = *$ and $\mathsf{GetTsk}(\rho, F) = S_1$, we have

$$\rho' = \mathsf{ClrTsk}((([ED], D, \mathsf{NTK}), ([BA], A, \mathsf{MAIN})), F) = (([F], D, \mathsf{NTK}), ([BA], A, \mathsf{MAIN})),$$

The aforementioned informal description of intent flags and the example provide the intuition of the semantics of $\mathsf{AMASS}_{ACT,IF}$. In the sequel, we formally define the semantics of $\mathcal{M}$ as a transition relation $\rho \xrightarrow[\tau]{\mathcal{M}} \rho'$ with $\tau = A \xrightarrow{\mathsf{start}(\phi)} B$.

Let us assume $\phi \models \neg\mathsf{TOH}$ first.

$\boxed{\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM}}$

- If $\phi \models \mathsf{CTP}$ and $B \in \mathsf{TopTsk}(\rho)$, then $\rho' = \mathsf{ClrTop}(\rho, B)$.
- If $\phi \models \mathsf{CTP}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $\rho' = \mathsf{Push}(\rho, B)$.
- If $\phi \models \neg\mathsf{CTP}$, then
  - if $\phi \models \mathsf{RTF}$ and $B \in \mathsf{TopTsk}(\rho)$, then $\rho' = \mathsf{MvAct2Top}(\rho, B)$,
  - if $\phi \models \mathsf{RTF}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $\rho' = \mathsf{Push}(\rho, B)$,
  - if $\phi \models \neg\mathsf{RTF}$, then
    * if $\phi \models \mathsf{STP}$ and $\mathsf{TopAct}(\rho) = B$ or $\phi \models \mathsf{STP} \wedge \mathsf{PIT}$ and $\mathsf{PreAct}(\rho) = B$, then $\rho' = \rho$,
    * otherwise, $\rho' = \mathsf{Push}(\rho, B)$.

$\boxed{\phi \models \mathsf{NDM}}$

- If $\phi \models \mathsf{MTK}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NDM})$.
- If $\phi \models \neg\mathsf{MTK}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$, then
    * if $\phi \models \mathsf{CTK}$, then $\rho' = \mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B)$,
    * if $\phi \models \neg\mathsf{CTK}$, then
      · if $B \in S_i$, then $\rho' = \mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B)$,
      · if $B \notin S_i$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
  - if $\mathsf{GetRealTsk}(\rho, B) = *$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NDM})$.

$\boxed{\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM}}$

- If $\phi \models \mathsf{MTK}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NTK})$.
- If $\phi \models \neg\mathsf{MTK}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$, then
    * if $\phi \models \mathsf{CTK}$, then $\rho' = \mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B)$,
    * if $\phi \models \neg\mathsf{CTK}$, then
      · if $\phi \models \mathsf{CTP}$ and $B \in S_i$, then $\rho' = \mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B)$,

    · if $\phi \models$ CTP and $B \notin S_i$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
    · if $\phi \models \neg$CTP, then
        ⋄ if $\phi \models$ RTF and $B \in S_i$, then $\rho' = \mathsf{MvAct2Top}(\mathsf{MvTsk2Top}(\rho, i), B)$,
        ⋄ if $\phi \models$ RTF and $B \notin S_i$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
        ⋄ if $\phi \models \neg$RTF, then
            ∘ if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq$ MAIN, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
            ∘ otherwise,
                ⋆ if $\phi \models$ STP and $\mathsf{TopAct}(S_i) = B$, or $\phi \models$ STP $\wedge$ PIT and $i = 1$ and $\mathsf{PreAct}(S_1) = B$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
                ⋆ otherwise, $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
- if $\mathsf{GetTsk}(\rho, B) = *$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NTK})$.

Finally we consider the situation $\phi \models$ TOH. Let ϕ' be obtained from ϕ by replacing TOH with $\neg$TOH. Moreover, suppose $\rho \xrightarrow[\tau']{\mathcal{M}} \rho'$, where $\tau' = A \xrightarrow{\mathsf{start}(\phi')} B$ and $\rho' = (\Omega_1, \cdots, \Omega_n)$. Then

- if $\phi \models$ NTK $\vee$ NDM, then let $\rho'' = (\Omega_1)$ and we have $\rho \xrightarrow[\tau]{\mathcal{M}} \rho''$,
- otherwise, $\rho \xrightarrow[\tau]{\mathcal{M}} \rho'$.

By a retrospect of the aforementioned formal semantics, we can discover that the intent flags may interfere with each other. We summarize their dependencies as follows.

Dependencies of intent flags. The dependencies of intent flags can exhibit in the following two forms: (1) n *subsumes* n', i.e., n' is ignored if n co-occurs with n', and the "subsume" relations are transitive, (2) n *enables* n', i.e., n' takes effect if n co-occurs with n'. The dependencies among the intent flags are summarized in Figure 5, where the solid lines represent the "subsume" relation, the dashed lines represent the "enable" relation.

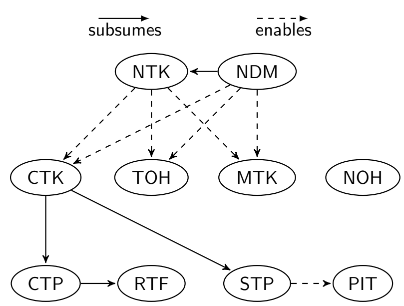

Fig. 5. Dependency graph of intent flags.

4.1.2 *Semantics of AMASS$_{FRG}$.*

In this case, we assume that there is only one activity A_0 and consider the fragments as the atomic objects, hence we only consider the transaction rules back , $A \xrightarrow{\mu} T$ and $F \xrightarrow{\mu} T$, where T is of the form $(\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))$ such that for every $j \in [k]$, $\beta_j \in \{\mathsf{ADD}, \mathsf{REP}, \mathsf{REM}\}$, $F_j \in \mathsf{Frg}$, $i_j \in \mathbb{N}$, and x_j is a variable storing the identifiers of fragment instances.

Intuitions.

The intuitions of the transition rules $A \xrightarrow{\mu} T$ and $F \xrightarrow{\mu} T$ are explained in the sequel.

- $\mathsf{ADD}(F, i, x)$ denotes the action where a *new* instance of the fragment F is added to the container i of the current activity. Moreover, the identifier of the new instance is stored in x.
- $\mathsf{REP}(F, i, x)$ is the same as $\mathsf{ADD}(F, i, x)$, except that the container i is cleared before adding F.
- $\mathsf{REM}(F, i, x)$ denotes the action where the instance of the fragment F of the identifier (stored in) x is removed from the container i of the current activity.
- $\mathsf{TS}[(\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))]$ denotes that the actions $\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k)$ are executed sequentially, and are added to the transaction stack.
- $\mathsf{NTS}[(\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))]$ is the same as TS, except that the transactions will *not* be added to the transaction stack.
  For convenience, these transition rules are called TS-transition rules and NTS-transition rules respectively.

It is fair to say that the intricacy of the semantics lies in the action back. Let A be the top activity of the top task, $\mathsf{Ctn}(A) = (i_1, \cdots, i_m)$, $V_1, \cdots, V_m$ be the fragment stacks for the containers $i_1, \cdots, i_m$, and $\eta \in \mathcal{T}^*$ be the transaction stack. If $\eta = \epsilon$, the activity A will be popped, otherwise the behavior of the action back is controlled by η, namely, when the back button is pressed, the top transaction of η is removed from η and its effects on the fragment stacks are to be revoked. Noticeable difficulties arise.

Mismatch between fragment and transaction stack. The transaction stack η may *not* contain all the historical transactions leading to the fragment stacks, but only those corresponding to the TS-transition rules, which means that when revoking the top action of η, say $\mathsf{ADD}(F, i_j, x_j)$, F may not be the top fragment of V_j. To deal with this issue, we

(1) store (F, n) in V_j, where n is the identifier of the instance of F which is generated when applying $\mathsf{ADD}(F, i_j, x_j)$,
(2) store the concretized action $\mathsf{ADD}(F, i_j, n)$, instead of $\mathsf{ADD}(F, i_j, x_j)$, into the transaction stack η.

As a result, we can revoke $\mathsf{ADD}(F, i_j, n)$ instead of $\mathsf{ADD}(F, i_j, x_j)$ in the transaction stack, and use n to identify the fragment instance to be removed from the fragment container V_j.

Revoking REP *actions.* The top transaction of η may include some action $\mathsf{REP}(F, i_j, n)$ and revoking these actions requires restoring the historical content of the container i_j before applying this action.

To deal with this issue, when an action $\mathsf{REP}(F, i_j, x_j)$ appears in a TS-transition rule and is to be pushed into the transaction stack, suppose that the current content of the container i_j is $((F'_1, n'_1), \cdots, (F'_k, n'_k))$, then $\mathsf{REP}(F, i_j, x_j)$ is concretized into the action sequence

$$\mathsf{REM}(F'_1, i_j, n'_1), \cdots, \mathsf{REM}(F'_k, i_j, n'_k), \mathsf{ADD}(F, i_j, n),$$

which—instead of $\mathsf{REP}(F, i_j, n)$—is pushed into the transaction stack, possibly together with the other actions in the same transaction. (Note that n is the identifier of the newly created instance of F.) Hence revoking $\mathsf{REP}(F, i_j, x_j)$ is to apply the concretized actions $\mathsf{REM}(F, i_j, n), \mathsf{ADD}(F'_k, i_j, n'_k), \cdots, \mathsf{ADD}(F'_1, i_j, n'_1)$ one by one.

Therefore, only concretized actions of the form $\mathsf{ADD}(F, i, n)$ or $\mathsf{REM}(F, i, n)$ are stored in the transaction stack, and we use $\mathcal{CT}$ to denote the set of these concretized actions.

Fragment containers and configurations.

A *fragment container* is encoded as

$$V = ((F_1, n_1), \cdots, (F_k, n_k)) \in (\mathsf{Frg} \times \mathbb{N})^+,$$

where n_j is the identifier of an instance of F_j for $j \in [k]$ and k is called the *height* of V.

A *configuration* ρ is encoded as a tuple (υ, η, ι), where $\upsilon = (V_1, \cdots, V_m)$ is a sequence of fragment containers associated with A_0, where $\mathsf{Ctn}(A_0) = (i_1, \cdots, i_m)$, $\eta = (T_1, \cdots, T_n) \in \mathcal{CT}^*$ is the transaction stack, and ι is the assignment function that assigns to each variable x in $\mathcal{M}$ an identifier, i.e. a natural number. Note that it is possible that $m = 0$ and/or $n = 0$. For technical convenience, let ι_0 denote the assignment function that assigns each variable x the value 0. The initial configuration of $\mathcal{M}$ is $((\epsilon, \cdots, \epsilon), \epsilon, \iota_0)$.

Auxiliary functions and predicates.

To specify the transition relation precisely and concisely, we define the following functions and predicates. For a container $V = ((F_1, n_1), \cdots, (F_m, n_m))$, define $\mathsf{TopFrg}(V) = F_1$.

Let $\rho = (\upsilon, \eta, \iota)$ be the configuration of associated with A_0, where $\mathsf{Ctn}(A_0) = (i_1, \cdots, i_k)$ $(k > 0)$, $\upsilon = (V_1, \cdots, V_k)$ is the container sequence, and $\eta = (T_1, \cdots, T_l)$ $(l \geq 0)$ is the transaction stack. Moreover, let $F \in \mathsf{Frg}$, $j \in [k]$, and x be a variable. We define the following functions.

- $\mathsf{TopFrg}(\upsilon) = \{\mathsf{TopFrg}(V_i) \mid i \in [k], V_i \neq \epsilon\}$ returns the set of topmost fragments of containers in υ.
- $\mathsf{ADD}(F, i_j, x)(\upsilon, \iota) = (\upsilon', \iota')$, where $\upsilon' = (V_1, \cdots, V_{j-1}, (F, n) \cdot V_j, V_{j+1}, \cdots, V_k)$, $\iota' = \iota[n/x]$, and n is the *minimum* identifier not occurring in υ or ι. Intuitively, $\mathsf{ADD}(F, i_j, x)(\upsilon, \iota)$ updates (υ, ι) by choosing a fresh identifier n, pushing (F, n) into the container i_j, and storing n into x.
- $\mathsf{REP}(F, i_j, x)(\upsilon, \iota) = (\upsilon', \iota')$, where $\upsilon' = (V_1, \cdots, V_{j-1}, (F, n), V_{j+1}, \cdots, V_k)$, $\iota' = \iota[n/x]$, and n is the *minimum* identifier not occurring in υ or ι. Intuitively, $\mathsf{REP}(F, i_j, x)(\upsilon, \iota)$ updates υ by replacing the content of container i_j with (F, n), and storing n into x.
- Suppose $V_j = ((F_1, n_1), \cdots, (F_m, n_m))$, then $\mathsf{REM}(F, i_j, x)(\upsilon, \iota) = (\upsilon', \iota')$, where
  - $\upsilon' = (V_1, \cdots, V_{j-1}, \tilde{V}, V_{j+1}, \cdots, V_k)$ such that
    - $\tilde{V} = V_j$, if $\iota(x) \neq n_{j'}$ for every $j' \in [m]$, and
    - $\tilde{V} = ((F_1, n_1), \ldots, (F_{l-1}, n_{l-1}), (F_{l+1}, n_{l+1}), \ldots, (F_m, n_m))$, if $\iota(x) = n_l$,
  - moreover, $\iota' = \iota$.

  Intuitively, the action $\mathsf{REM}(F, i_j, x)(\upsilon)$ updates υ by removing the instance of F of the identifier $\iota(x)$ from container i_j and does not change ι.
- Furthermore, the functions $\mathsf{ADD}(F, i_j, n)(\upsilon, \iota)$, $\mathsf{REP}(F, i_j, n)(\upsilon, \iota)$, $\mathsf{REM}(F, i_j, n)(\upsilon, \iota)$ for concretized actions can be defined similarly (except that ι is unchanged).

For a transaction $T = (\beta_1(F_1, j_1, x_1), \cdots, \beta_r(F_r, j_r, x_r))$ such that $\beta_s \in \{\mathsf{ADD}, \mathsf{REM}, \mathsf{REP}\}$ and $F_s \in \mathsf{Frg}$ for every $s \in [r]$, $\mathsf{UpdateCtns}_T(\upsilon, \iota) = (\upsilon_r, \iota_r)$, where $(\upsilon_0, \iota_0) = (\upsilon, \iota)$, and for every $s \in [r]$, $(\upsilon_s, \iota_s) = \beta_s(F_s, j_s, x_s)(\upsilon_{s-1}, \iota_{s-1})$, i.e. $\mathsf{UpdateCtns}_T$ updates the containers and the assignment function by applying the actions in T. Furthermore, $\mathsf{UpdateCtns}_T(\upsilon, \iota)$ can be defined similarly for concretized transactions T.

We also introduce a function that concretize the actions REP by utilizing the containers in υ. Suppose $F \in \mathsf{Frg}$, $j \in [k]$, x is a variable, and $V_j = ((F_1, n_1), \cdots, (F_m, n_m))$. Let n be the minimum identifier not occurring in υ or ι. Then

$$\mathsf{Concretize}_{\upsilon,\iota}(\mathsf{REP}(F, i_j, x)) = \mathsf{REM}(F_1, i_j, n_1), \cdots, \mathsf{REM}(F_m, i_j, n_m), \mathsf{ADD}(F, i_j, n).$$

Moreover, let

$$\mathsf{Concretize}_{\upsilon,\iota}(\mathsf{ADD}(F, i_j, x)) = \mathsf{ADD}(F, i_j, n) \text{ and } \mathsf{Concretize}_{\upsilon,\iota}(\mathsf{REM}(F, i_j, x)) = \mathsf{REM}(F, i_j, \iota(x))$$

by convention.

Let $T = (\beta_1(F_1, j_1, x_1), \cdots, \beta_k(F_r, j_r, x_r))$ be a transaction such that $\beta_s \in \{\mathsf{ADD}, \mathsf{REM}, \mathsf{REP}\}$, and $F_s \in \mathsf{Frg}$ for every $s \in [r]$. We define $\mathsf{Concretize}_{\upsilon,\iota}(T)$ as the concatenation of the concretized action sequences $\mathsf{Concretize}_{\upsilon_0,\iota_0}(\beta_1(F_1, j_1, x_1))$, $\cdots$, $\mathsf{Concretize}_{\upsilon_{s-1},\iota_{s-1}}(\beta_s(F_s, j_s, x_s))$, where $(\upsilon_0, \iota_0) = (\upsilon, \iota)$, and for each $s \in [r]$, $(\upsilon_s, \iota_s) = \beta_s(F_s, j_s, x_s)(\upsilon_{s-1}, \iota_{s-1})$.

Finally, we define functions $\beta^{-1}(F, i_j, n)$ for $\beta(F, i_j, n) \in \mathcal{CT}$ as follows.

- $\mathsf{ADD}^{-1}(F, i_j, n)(\upsilon, \iota) = \mathsf{REM}(F, i_j, n)(\upsilon, \iota)$,
- $\mathsf{REM}^{-1}(F, i_j, n)(\upsilon, \iota) = \mathsf{ADD}(F, i_j, n)(\upsilon, \iota)$.

Intuitively, $\mathsf{ADD}(F, i_j, n)$ and $\mathsf{REM}(F, i_j, n)$ are dual actions. Moreover, for a concretized transaction

$$T = (\beta_1(F_1, j_1, n_1), \cdots, \beta_r(F_r, j_r, n_r))$$

such that $\beta_s \in \{\mathsf{ADD}, \mathsf{REM}, \mathsf{REP}\}$, $F_s \in \mathsf{Frg}$ for every $s \in [r]$, and $n \in \mathbb{N}$, we define T^{-1} as

$$(\beta_r^{-1}(F_r, j_r, n_r), \cdots, \beta_1^{-1}(F_1, j_1, n_1)).$$

Transition relation.

We assume that the current activity is A. For $\rho \xrightarrow[\tau]{\mathcal{M}} \rho'$, we let $\rho = (\upsilon, \eta, \iota)$ be the current configuration, where $\mathsf{Ctn}(A) = (i_1, \cdots, i_k)$, $\upsilon = (V_1, \cdots, V_k)$, and $\eta = (T_1, \cdots, T_l)$.

- If $\tau = \mathsf{back}$, then we assume that $\eta \neq \epsilon$, otherwise A will be popped and there is no activity to store fragments. Then $\rho' = (\upsilon', \eta', \iota')$, where $(\upsilon', \iota') = \mathsf{UpdateCtns}_{T_1^{-1}}(\upsilon, \iota)$, and $\eta' = (T_2, \cdots, T_l)$. Note that T_1 is popped off the transaction stack and the actions of T_1 are revoked on (υ, ι).
- If $\tau = A \xrightarrow{\mu} T$, or $\tau = F \xrightarrow{\mu} T$, and $F \in \mathsf{TopFrag}(\upsilon)$ we let

  $$T = (\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k)),$$

  then $\rho' = (\upsilon', \eta', \iota')$ is obtained from ρ by applying all the actions in T to the containers and assignment function, such that $(\upsilon', \iota') = \mathsf{UpdateCtns}_T(\upsilon, \iota)$ and $\eta' = \mathsf{Concretize}_{\upsilon,\iota}(T) \cdot \eta$ if $\mu = \mathsf{TS}$, $\eta' = \eta$ otherwise. Note that the difference between TS and NTS is that the concretization of T is stored into the transaction stack when $\mu = \mathsf{TS}$.

We use the following example to illustrate the formal semantics of AMASS_{FRG}.

Example 4.3. Let $\mathcal{M} = (\mathsf{Act}, A_0, \mathsf{Frg}, \mathsf{Lmd}, \mathsf{Aft}, \mathsf{Ctn}, \Delta)$ be a AMASS_{FRG}, where $\mathsf{Act} = \{A_0\}$, $\mathsf{Frg} = \{F_1, F_2, F_3\}$, and $\mathsf{Ctn}(A_0) = (1)$. Moreover, $\Delta = \{\mathsf{back}\} \cup \{\tau_i \mid 0 \leq i \leq 3\}$, where $\tau_0 = \mathsf{back}$, $\tau_1 = F_1 \xrightarrow{\mathsf{TS}} (\mathsf{ADD}(F_2, 1, x))$, $\tau_2 = F_2 \xrightarrow{\mathsf{NTS}} (\mathsf{REP}(F_3, 1, x))$, and $\tau_3 = F_3 \xrightarrow{\mathsf{NTS}} (\mathsf{REM}(F_3, 1, x))$. Then the relation $\xrightarrow{\mathcal{M}}$ on the set of configurations that are reachable from the initial configuration $(([(F_1, 0)]), \epsilon, \{x = 0\})$ is illustrated in Figure 6.

For instance,

- from the configuration $(([(F_1, 0)]), \epsilon, \{x = 0\})$, after executing the transition rule $F_1 \xrightarrow{\mathsf{TS}} (\mathsf{ADD}(F_2, 1, x))$, we have

  $$(\upsilon', \iota') = \mathsf{UpdateCtns}_T(\upsilon, \iota) = \mathsf{ADD}(F_2, 1, x)([(F_1, 0)], \{x = 0\}) = ([(F_2, 1)(F_1, 0)], \{x = 1\}),$$

$$\eta' = \mathsf{Concretize}_{v,\iota}(T) \cdot \eta = \mathsf{Concretize}_{v,\iota}(\mathsf{ADD}(F_2, 1, x)) = \mathsf{ADD}(F_2, 1, 1),$$

- from the configuration $(([(F_2, 1), (F_1, 0)]), [(\mathsf{ADD}(F_2, 1, 1))], \{x = 1\})$, after executing the transition rule $F_2 \xrightarrow{\mathsf{NTS}} (\mathsf{REP}(F_3, 1, x))$, we have $\eta' = \eta$ and

$$(v', \iota') = \mathsf{UpdateCtns}_T(v, \iota) = \mathsf{REP}(F_3, 1, x)([(F_2, 1), (F_1, 0)], \{x = 1\}) = ([(F_3, 2)], \{x = 2\}),$$

- from the configuration $(([(F_2, 1), (F_1, 0)]), [(\mathsf{ADD}(F_2, 1, 1))], \{x = 1\})$, after executing the transition rule back, we have $T_1 = \mathsf{ADD}(F_2, 1, 1)$ and $T_1^{-1} = \mathsf{REM}(F_2, 1, 1)$, $\eta' = \epsilon$, moreover,

$$(v', \iota') = \mathsf{UpdateCtns}_{T_1^{-1}}(v, \iota) = \mathsf{REM}(F_2, 1, 1)([(F_2, 1), (F_1, 0)], \{x = 1\}) = ([(F_1, 0)], \{x = 1\}),$$

- from the configuration $(([(F_3, 2)]), [(\mathsf{ADD}(F_2, 1, 1))], \{x = 2\})$, after executing the transition rule $F_3 \xrightarrow{\mathsf{NTS}} (\mathsf{REM}(F_3, 1, x))$, we have $\eta' = \eta$, moreover

$$(v', \iota') = \mathsf{UpdateCtns}_T(v, \iota) = \mathsf{REM}(F_3, 1, 2)([(F_3, 2)], \{x = 2\}) = ([\epsilon], \{x = 2\}),$$

- from the configuration $(([(F_3, 2)]), [(\mathsf{ADD}(F_2, 1, 1))], \{x = 2\})$, after executing the transition rule back, we have $T_1 = \mathsf{ADD}(F_2, 1, 1)$ and $T_1^{-1} = \mathsf{REM}(F_2, 1, 1)$, $\eta' = \epsilon$, moreover,

$$(v', \iota') = \mathsf{UpdateCtns}_T(v, \iota) = \mathsf{REM}(F_2, 1, 1)([(F_3, 2)], \{x = 2\}) = ([(F_3, 2)], \{x = 2\}).$$

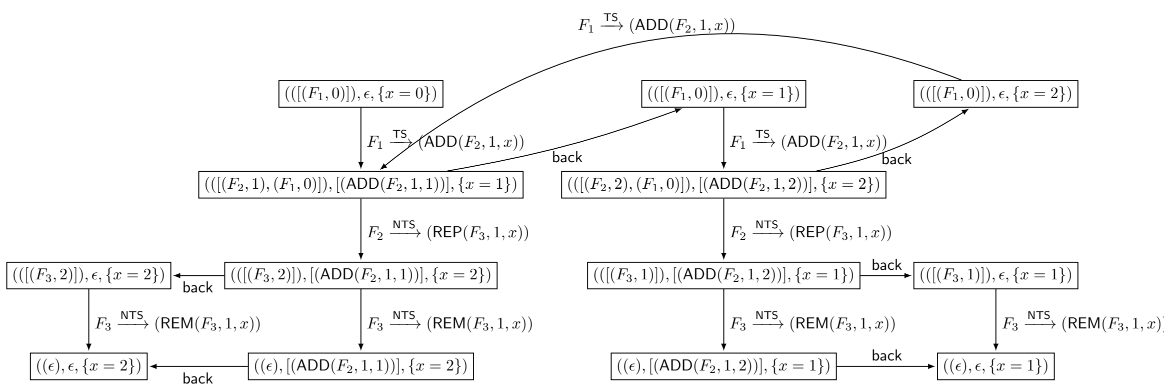

Fig. 6. Configurations that are reachable from the configuration $(([(F_1, 0)]), \epsilon, \{x = 0\})$ in $\mathcal{M}$

4.2 Semantics of AMASS$_{ACT}$ for the other versions of Android

As mentioned before, the semantics of AMASS$_{ACT}$ of different versions of Android are different. In the sequel, we illustrate the differences by focusing on AMASS$_{ACT,IF}$. The differences of the semantics of AMASS$_{ACT}$ can be found in the appendix.

Before a formal description, let us first illustrate the differences using the following example.

Example 4.4. Let $\mathcal{M} = (\mathsf{Act}, A, \mathsf{Frg}, \mathsf{Lmd}, \mathsf{Aft}, \mathsf{Ctn}, \Delta)$ be an AMASS$_{ACT,IF}$, where $\mathsf{Act} = \{A, B, C, D\}$, for each $A' \in \mathsf{Act}$, $\mathsf{Lmd}(A') = \mathsf{STD}$, $\mathsf{Aft}(A) = \mathsf{Aft}(B) = 1$, and $\mathsf{Aft}(C) = \mathsf{Aft}(D) = 2$, moreover, $\Delta = \{\mathsf{back}\} \cup \{\tau_i \mid 1 \le i \le 6\} \cup \{\tau_i' \mid 1 \le i \le$

3} such that $\tau_1 = A \xrightarrow{\text{start}(\bot)} C$, $\tau_2 = C \xrightarrow{\text{start}(\text{NTK}\wedge\text{MTK})} B$, $\tau_3 = B \xrightarrow{\text{start}(\text{NTK})} C$, $\tau_4 = C \xrightarrow{\text{start}(\bot)} D$, $\tau_5 = D \xrightarrow{\text{start}(\bot)} A$, $\tau_6 = A \xrightarrow{\text{start}(\text{NTK})} A$, $\tau_1' = C \xrightarrow{\text{start}(\text{NTK}\wedge\text{RTF})} D$, $\tau_2' = C \xrightarrow{\text{start}(\text{RTF})} A$, $\tau_3' = C \xrightarrow{\text{start}(\text{NTK})} B$.

We use the configuration $(([CA], A, \text{MAIN}), ([ADC], C, \text{NTK}), ([B], B, \text{NTK}))$ to illustrate the difference. This configuration can be reached from the initial configuration $(([A], A, \text{MAIN}))$ by the following sequence of transition rules: $\tau_1\ \tau_2\ \tau_3\ \tau_4\ \tau_5\ \tau_6$ back. The differences of semantics for different versions of Android are demonstrated by applying $\tau_1', \tau_2', \tau_3'$ on the configuration $(([CA], A, \text{MAIN}), ([ADC], C, \text{NTK}), ([B], B, \text{NTK}))$ (see Figure 7).

- If the transition $\tau_1' = C \xrightarrow{\text{start}(\text{NTK}\wedge\text{RTF})} D$ is applied, then for Android 11.0-13.0 the task $([ADC], C, \text{NTK})$ is moved to the top and the activity D is reordered to the front, resulting in the configuration

  $$(([DAC], C, \text{NTK}), ([CA], A, \text{MAIN}), ([B], B, \text{NTK})),$$

  while for Android 6.0-10.0, the activity D is pushed after the task $([ADC], C, \text{NTK})$ is moved to the top, resulting in the configuration

  $$(([DADC], C, \text{NTK}), ([CA], A, \text{MAIN}), ([B], B, \text{NTK})).$$

  This difference is explained by the fact that for Android 6.0-10.0, RTF is ignored when it is used together with NTK.
- If the transition $\tau_2' = C \xrightarrow{\text{start}(\text{RTF})} A$ is applied, then for all the versions of Android 6.0–13.0, except 7.0, A is reordered to the front in the top task, resulting in the configuration

  $$(([AC], A, \text{MAIN}), ([ADC], C, \text{NTK}), ([B], B, \text{NTK})).$$

  In Android 7.0, the top task is cleared and A is pushed, resulting in the configuration

  $$(([A], A, \text{MAIN}), ([ADC], C, \text{NTK}), ([B], B, \text{NTK})).$$

  This difference is explained by the fact that for Android 7.0, when the top task is the main task where the started activity occurs but is not the top activity, RTF has the same effect as CTK.
- If the transition $\tau_3' = C \xrightarrow{\text{start}(\text{NTK})} B$ is applied, then for all the versions of Android 7.0–12.0, the B-task is moved to the top (without pushing a new instance of B), resulting in the configuration

  $$(([B], B, \text{NTK}), ([CA], A, \text{MAIN}), ([ADC], C, \text{NTK})).$$

  In Android 6.0, the B-task is not moved to the top, and an instance of B is pushed to the top task directly, resulting in the configuration

  $$(([BCA], A, \text{MAIN}), ([ADC], C, \text{NTK}), ([B], B, \text{NTK})).$$

  This difference is explained by the fact that the task allocation mechanism of Android 6.0 is different from the other versions, namely, in Android 6.0, only affinities are used for looking for a task, while for the other versions, real activities and affinities are used together to look for a task.

In the sequel, we state the differences of the semantics of $\text{AMASS}_{ACT,IF}$ models in details. To avoid tediousness, let us focus on the situation $\phi \models \neg\text{TOH}$. The differences for the situation $\phi \models \text{TOH}$ are similar.

4.2.1 *Android 11.0, 12.0.*

The semantics of AMASS for Android 11.0 and 12.0 are the same as Android 13.0.

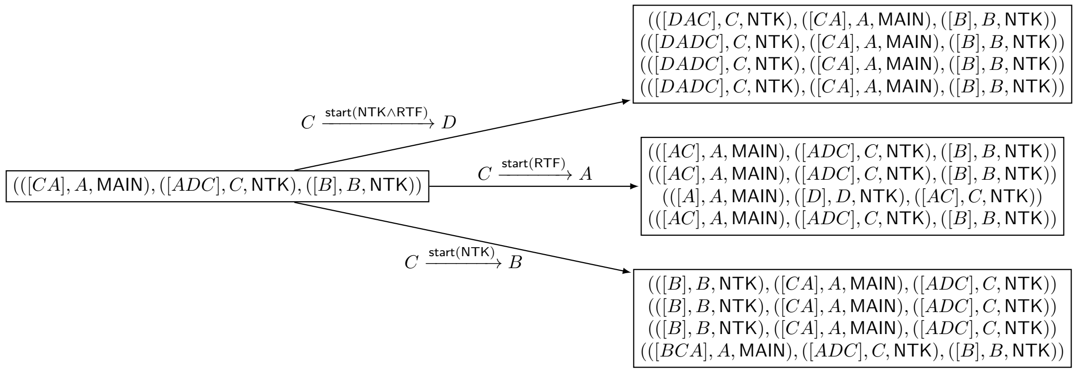

Fig. 7. Semantics of AMASS$_{ACT,IF}$ for different versions of Android, where the 1st, 2nd, 3rd, 4th line of the texts in the right boxes are configurations for Android 11.0–13.0, 8.0–10.0, 7.0, and 6.0 respectively

4.2.2 Android 10.0, 9.0, and 8.0. The semantics for these three versions are the same and differ from that for Android 13.0 in the following sense: RTF is ignored when used together with NTK. That is, for Android 10.0, 9.0, and 8.0, the semantics of AMASS$_{ACT,IF}$ for the case $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM}$ is adapted from Android 13.0 as follows.

- If $\phi \models \mathsf{MTK}$, then $\cdots$.
- If $\phi \models \neg\mathsf{MTK}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$, then
    - if $\phi \models \mathsf{CTK}$, then $\cdots$,
    - if $\phi \models \neg\mathsf{CTK}$, then
      - if $\phi \models \mathsf{CTP}$ and $B \in S_i$, then $\cdots$,
      - if $\phi \models \mathsf{CTP}$ and $B \notin S_i$, then $\cdots$,
      - if $\phi \models \neg\mathsf{CTP}$, then
        - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq \mathsf{MAIN}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
        - otherwise,
          - if $\phi \models \mathsf{STP}$ and $\mathsf{TopAct}(S_i) = B$, or $\phi \models \mathsf{STP} \wedge \mathsf{PIT}$ and $i = 1$ and $\mathsf{PreAct}(S_1) = B$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
          - otherwise, $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
  - if $\mathsf{GetTsk}(\rho, B) = *$, then $\cdots$.

Note that the parts of the semantics denoted by $\cdots$ are the same as Android 13.0, and in the semantics for the situation $\phi \models \neg\mathsf{CTP}$, the flag RTF has no effects, thus is ignored.

4.2.3 Android 7.0. The semantics for Android 7.0 is close to that of Android 10.0 (or 9.0, 8.0) but differs from it in the following two aspects: 1) the effect of NDM is the same as that of NTK, 2) when $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{CTP}$, if the top task is the main task where the started activity occurs but is not the top activity, then RTF has the same effect as CTK. More precisely, for Android 7.0, only two cases $\phi \models \mathsf{NTK}$ and $\phi \models \neg\mathsf{NTK}$ are considered, where the semantics for the case $\phi \models \mathsf{NTK}$ inherits that of $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM}$ for Android 10.0, while the semantics for the case $\phi \models \neg\mathsf{NTK}$ is adapted from that of $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM}$ for Android 10.0 as follows.

- If $\phi \models \mathsf{CTP}$ and $B \in \mathsf{TopTsk}(\rho)$, then $\cdots$.

- If $\phi \models \mathsf{CTP}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $\cdots$.
- If $\phi \models \neg\mathsf{CTP}$, then
  - if $\phi \models \mathsf{RTF}$ and $B \in \mathsf{TopTsk}(\rho)$, then
    - if $\zeta_i = \mathsf{MAIN}$, then $\rho' = \mathsf{ClrTsk}(\rho, B)$,
    - otherwise, $\rho' = \mathsf{MvAct2Top}(\rho, B)$,
  - if $\phi \models \mathsf{RTF}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $\cdots$,
  - if $\phi \models \neg\mathsf{RTF}$, then $\cdots$.

4.2.4 Android 6.0. The semantics for Android 6.0 differs from that of Android 10.0 (or 9.0, 8.0) in the following two aspects: 1) the effect of NDM is the same as that of NTK, 2) the task allocation mechanism of Android 6.0 does not use the real activities of tasks and only relies on affinities. More precisely, for Android 6.0, only two cases $\phi \models \mathsf{NTK}$ and $\phi \models \neg\mathsf{NTK}$ are considered, where the semantics for the case $\phi \models \neg\mathsf{NTK}$ inherits that of $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM}$, and the semantics for the case $\phi \models \mathsf{NTK}$ is adapted from that of $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM}$ for Android 10.0 as follows, where the conditions involving $\mathsf{GetRealTsk}(\rho, B)$ and $\mathsf{GetTsk}(\rho, B)$ are simplified into the conditions involving only $\mathsf{GetTsk}(\rho, B)$, moreover, we do not need to distinguish whether a task is the main task or not.

- If $\phi \models \mathsf{MTK}$, then $\cdots$.
- If $\phi \models \neg\mathsf{MTK}$, then
  - if $\mathsf{GetTsk}(\rho, B) = S_i$, then
    - if $\phi \models \mathsf{CTK}$, then $\cdots$,
    - if $\phi \models \neg\mathsf{CTK}$, then
      - if $\phi \models \mathsf{CTP}$ and $B \in S_i$, then $\cdots$,
      - if $\phi \models \mathsf{CTP}$ and $B \notin S_i$, then $\cdots$,
      - if $\phi \models \neg\mathsf{CTP}$, then
        - if $\phi \models \mathsf{STP}$ and $\mathsf{TopAct}(S_i) = B$, or $\phi \models \mathsf{STP} \wedge \mathsf{PIT}$ and $i = 1$ and $\mathsf{PreAct}(S_1) = B$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
        - otherwise, $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
  - if $\mathsf{GetTsk}(\rho, B) = *$, then $\cdots$.

We remark that the unexpected behavior of RTF for Android 7.0, that is, RTF behaves the same as CTK in some cases, does not occur in Android 6.0.

We have defined the semantics of AMASS in this section. To go one step further, we would like to validate the semantics against the actual behaviors of Android apps w.r.t. the multitasking mechanism between activities and fragments. Moreover, we would like to do some static analysis of the behaviors of Android apps related to the multitasking mechanism between activities and fragments. To facilitate these two tasks, in the next two sections, we build AMASS models out of APK files of Android apps and encode the reachability problem of AMASS models into the model checking problem in the input format of nuXmv model checker [CCD+14a].

5 GENERATING AMASS MODELS FROM APK FILES

The goal of this section is to show how we build AMASS models out of the APK files of the Android apps.

In Android apps, switching between components is by sending inter-component communication (ICC) messages to the Android system. Each ICC message is represented by an `Intent` object that carries a set of data items. By resolving the ICC messages, we can extract both the source and destination of the switching and the data items it carries.

The recent work [YZL+22a] studied the available ICC resolution tools and performed a practical evaluation on both hand-crafted benchmarks as well as large-scale real-world apps. The evaluation shows that (1) the fragment is a key feature that is widely used in the component transitions; (2) most of state-of-the-art ICC resolution tools miss the ICCs involving activities and fragments. A recent tool *ICCBot* [YZL+22b] resolves the component transitions connected by fragments and activities, achieves a higher success rate and accuracy. However, ICCBot is insufficient for obtaining an AMASS model for the following two reasons.

- ICCBot fails to decompile the APK files of some commercial apps, thus their models cannot be constructed.
- AMASS models contain more information about the component transitions, in particular, ICCBot ignores the API "addToBackStack()" which can allow pushing transactions to the fragment transaction stack, and ignores the API `finish()` which can pop the current activity.

To this end, we extend ICCBot to $\text{ICCBot}_{\text{AMASS}}$ in the following aspects.

- At first, $\text{ICCBot}_{\text{AMASS}}$ records the calls to the API `finish()` and associates them with the corresponding calls to `startActivity()` or `startActivityForResult()`. This enables $\text{ICCBot}_{\text{AMASS}}$ to accurately track the termination of the activities and identify transition rules that are affected by these terminations.
- Additionally, $\text{ICCBot}_{\text{AMASS}}$ monitors the calls of the API `addToBackstack()`, so that the transition rules involving fragments can be accurately constructed. In contrast, ICCBot does not record `addToBackstack()`. Therefore, it misses the information $\mu \in \{\mathsf{TS}, \mathsf{NTS}\}$ in the transition rules $A \xrightarrow{\mu} T$ or $F \xrightarrow{\mu} T$.
- Moreover, $\text{ICCBot}_{\text{AMASS}}$ provides means to take cross-app ICCs into consideration when constructing AMASS models.
- Finally, for the commercial apps that cannot be decompiled, $\text{ICCBot}_{\text{AMASS}}$ applies a dynamic approach to extract the AMASS models.

In the sequel, we describe more details about how $\text{ICCBot}_{\text{AMASS}}$ takes cross-app ICCs into consideration as well as utilizes a dynamic approach to construct AMASS models for these apps that cannot be decompiled.

Model extraction for cross-app ICCs. If for an app, the multiple APK files that are involved in the cross-app ICCs of this app are provided, then $\text{ICCBot}_{\text{AMASS}}$ can construct an AMASS model that captures the behaviors of all these multiple APK files. The algorithm goes as follows.

(1) At first, for the APK file corresponding to an app, $\text{ICCBot}_{\text{AMASS}}$ constructs an AMASS model, where the cross-app ICCs starting from this app are recorded but not processed.
(2) Then for the model, we try to manually identify the APK files involved in these cross-app ICCs by searching for their package names.
(3) If the APK files involved in the cross-app ICCs are identified successfully for the app, then all these multiple APK files, including the original APK file for the app, are processed and an AMASS model to capture the behaviors of all these apps is constructed.

Afterwards, these models can be analyzed just as a model extracted from an app containing no cross-app ICCs.

Dynamic model extraction for unsuccessfully decompiled APK files. Roughly speaking, the algorithm to extract AMASS models dynamically from APK files works as follows.

(1) We first execute all click events of the foreground activity to search for new activities.

(2) When clicking some events in the foreground activity, some new UI widgets, e.g. menu widgets, may appear (though the foreground activity is unchanged), resulting in some new click events. When executing these newly produced click events, some UI widgets may appear and even more new click events can be produced. The click events in the foreground activity are organized hierarchically.

(3) We apply a depth-first search to execute all the sequences of hierarchically-organized click events, up to some depth. If a new activity is found, the associated transition rule for starting the new activity is produced, where the intent flags are guessed. For instance, if a non-top task is moved to the top and a new instance of the activity B of the "standard" launch mode is pushed to it, then the "NTK" flag can be guessed.

Algorithm 1: *AMASSExplore*()

```
/* Two global variables: Δ and acts */;
ρ ← adb.getTasks();
topAct ← adb.getTopActivity();
visitedActs ← {topAct};
DynDFS(ρ, topAct, [], d);
while acts ≠ ∅ do
    foreach (newAct, newEvents) ∈ acts do
        if newAct ∈ visitedActs then
            acts ← acts − {(newAct, newEvents)};
            continue;
        else
            visitedActs ← visitedActs ∪ {newAct};
            UIAutomator.restart();
            foreach event ∈ newEvents do
                UIAutomator.click(event);
            ρ ← ADB.getTasks();
            DynDFS(ρ, newAct, newEvents, d);
```

The pseudo-code of the dynamic model-extraction algorithm is given in Algorithm 1, namely, the procedure $AMASSExplore()$. The procedure utilizes two global variables, Δ and $acts$, where Δ represents the current set of transition rules that are discovered so far, and $acts$ stores the pairs $(A, events)$, where A is an activity and $events$ is a sequence of click events so that from the main activity, if these click events are executed by UIAutomator, then an instance of A is started. The procedure $AMASSExplore()$ calls the procedure $DynDFS(\rho, srcAct, events, d)$ (see Algorithm 2) to do a depth-first search for new activities, starting from $srcAct$, up to the depth d (i.e. the length of the click events reaching a new activity). The two procedures relies on some APIs of UIAutomator and ADB, whose meanings are described below.

- $UIAutomator.getClickEvents()$ returns the set of clickable events in the foreground activity of the app under test,
- $UIAutomator.click(e)$ executes the click-event e,
- $UIAutomator.restart()$ restarts the the app under test,
- $ADB.getTasks()$ returns the content of the back stack.

For each $(newAct, newEvents) \in acts$, $AMASSExplore()$ calls the procedure $DynDFS(\rho, newAct, newEvents, d)$ to search dynamically for new activities, starting from $newAct$, and store the newly discovered activities as well as the corresponding sequences of click events into *acts*, up to the depth d. Here d represents the length of the click-events sequence required to reach the new activity from *newAct*. Note that *newEvents* are used by UIAutomator to reach *newAct* starting from the main activity. In some cases, pressing buttons may not start a new activity, but instead, new widgets appear (e.g., the menu widget), requiring us to continue searching for new click-events in these widgets to reach the new activity.

Algorithm 2: $DynDFS(\rho, srcAct, events, d)$

```
input  : ρ, srcAct, events, d
/* Dynamic depth-first search new activity with UIAutomator, up to depth d */;
if d > 0 then
    foreach event ∈ UIAutomator.getClickEvents() do
        UIAutomator.click(event);
        newAct ← ADB.getTopActivity();
        newEvents ← events · event;
        if newAct = srcAct then
            if d > 1 then
                DynDFS(ρ, srcAct, newEvents, d − 1);
                /* Decrement d and continue the dynamic DFS. */;
            else
                UIAutomator.restart();
                foreach event ∈ events do
                    UIAutomator.click(event);
        else
            /* Discover new transitions and refine the AMASS model */;
            ρ′ ← ADB.getTasks();
            τ′ ← guessTrans(srcAct, ρ, newAct, ρ′);
            Δ ← Δ ∪ {τ′};
            acts ← acts ∪ {(newAct, newEvents)};
            UIAutomator.restart();
            foreach event ∈ events do
                UIAutomator.click(event);
```

If a new activity is found during the execution of the procedure $DynDFS(\rho, srcAct, Events, d)$, with ρ' being the current configuration, then the function $guessTrans(srcAct, \rho, newAct, \rho')$ is used to infer the intent flags based on the evolution of the configuration, so that a new transition rule τ' can be formed and put into Δ. Moreover, the sequence of click events that reach *newAct* from *srcAct* is recorded in *newEvents* and the pair $(newAct, newEvents)$ is put into *acts*. In order to continue searching for new activities, we need to return to the activity *srcAct*. Instead of pressing the back button, we restart the app and execute the corresponding *events* using UIAutomator. This is necessary because in some cases, pressing the back button may not allow us to reach the previous activity, as indicated by the semantics in Section 4.

The aforementioned dynamic model-extraction algorithm enables us to extract models from the APK files that cannot be decompiled successfully. Nevertheless, the models extracted dynamically may be inaccurate. Specifically, the following three types of inaccuracies may exist in the dynamically-extracted models.

- The intent flags guessed in the algorithm may be incorrect, since the effects of different combinations of intent flags may be equivalent.
- Some transition rules involving fragments may be missed, since fragments are ignored in the algorithm, due to the fact that it is challenging to identify dynamically the fragment to which an UI widget belongs.
- Furthermore, even the transition rules for activities may be incomplete, since it is typical that the dynamic exploration does not cover all the possible behaviors of Android apps.

Finally, compared to the static model extraction, the dynamic model extraction is slower in general, as demonstrated by the experiments in Section 9 (see Table 9-10 for details).

6 ENCODING AMASS MODELS INTO NUXMV

In this section, we show how to encode the semantics of these AMASS models into the reachability problem of finite state machines that can be tackled by the symbolic model checker nuXmv [CCD+14b]. This encoding will facilitate the automated validation of the formal semantics of AMASS models (Section 7) and static analysis of Android apps (Section 8).

In general, AMASS models are infinite-state systems, since both the number of tasks (stacks) and the heights of stacks can be unbounded. We impose two predefined upper bounds, i.e., c_t on the maximum number of tasks of the same affinity, and $\hbar$ on the height of stacks. As a result, we can obtain a finite state system. Nevertheless, even with these upper bounds, the number of configurations of an AMASS model can be *exponential* in the size of the model and c_t and $\hbar$, for which we resort to nuXmv [CCD+14b] and translate an AMASS $\mathcal{M}$ (with given c_t and $\hbar$) to an FSM $\mathcal{A}_{\mathcal{M}}$.

In nuXmv, an FSM $\mathcal{A}$ is triplet $(\vec{x}, \vec{s}, \delta)$, where

- $\vec{x}$ is a tuple of Boolean variables representing the inputs,
- $\vec{s}$ is a tuple of Boolean variables representing the states,
- δ is a Boolean formula involving the variables $\vec{s}$, $\vec{x}$, and $\vec{s'}$ to describe the transitions, where $\vec{s'}$ is a tuple of primed state variables that represent their new values after a transition.

Let $\mathcal{M} = (\mathsf{Act}, A_0, \mathsf{Frg}, \mathsf{Ctn}, \mathsf{Lmd}, \mathsf{Aft}, \Delta)$ be an AMASS and $\mathsf{Conf}_{\mathcal{M},c_t,\hbar}$ denote the set of configurations of $\mathcal{M}$ where the number of tasks of the same affinity is no more than c_t and the heights of all stacks are no more than $\hbar$. It follows that $(\mathsf{Conf}_{\mathcal{M},c_t,\hbar}, \xrightarrow{\mathcal{M}})$ is a finite state system.[7] Then we show how to encode $(\mathsf{Conf}_{\mathcal{M},c_t,\hbar}, \xrightarrow{\mathcal{M}})$ into an FSM $\mathcal{A}_{\mathcal{M}}$. Intuitively, the states of $\mathcal{A}_{\mathcal{M}}$ represent the configurations in $\mathsf{Conf}_{\mathcal{M},c_t,\hbar}$ and the transitions of $\mathcal{A}_{\mathcal{M}}$ simulate the transition rules of $\mathcal{M}$.

To facilitate the encoding, we first introduce the following definitions.

- Let k_c denote the maximum length of $\mathsf{Ctn}(A)$ for $A \in \mathsf{Act}$, that is, the maximum number of fragment containers in an activity.
- Let k_a denote the maximum number of actions occurring in the fragment transaction of one transition rule of $\mathcal{M}$.
- Let X denote the set of variables occurring in the transitions of $\mathcal{M}$ for storing the identifiers of fragment instances and $k_i = |X|$.
- Let $k_t = c_t|\mathsf{Aft}(\mathsf{Act})|$, namely, the maximum number of tasks in the task stacks.

[7] Strictly speaking, $\xrightarrow{\mathcal{M}}$ should be the restriction of $\xrightarrow{\mathcal{M}}$ to $\mathsf{Conf}_{\mathcal{M},c_t,\hbar}$

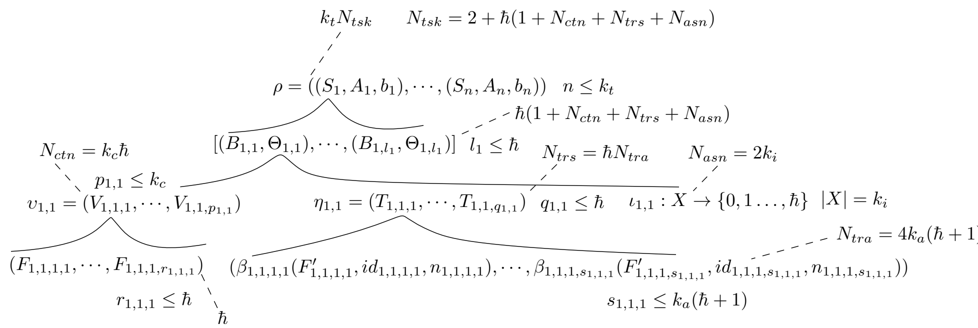

Fig. 8. Length of the word encoding a task stack

Recall that each configuration in $\mathrm{Conf}_{\mathcal{M}}$ is $\rho = ((S_1, A_1, b_1), \cdots, (S_n, A_n, b_n))$. Let

$$\Sigma_{\mathcal{M}} = \mathsf{Act} \cup \mathsf{Frg} \cup \{\mathsf{ADD}, \mathsf{REM}\} \cup \mathsf{CID}_{\mathcal{M}} \cup X \cup [k_c\hbar] \cup \{0, 1, \bot\},$$

where $\mathsf{CID}_{\mathcal{M}}$ is the set of fragment container identifiers occurring in $\mathcal{M}$ and $\bot$ is a dummy symbol introduced as placeholders. Then each task stack in $\mathrm{Conf}_{\mathcal{M},\hbar}$ can be encoded by a word of length *exactly* $k_t(2 + \hbar(1 + k_c\hbar + 4k_a\hbar(\hbar + 1) + k_i))$ over the alphabet $\Sigma_{\mathcal{M}}$. The arguments for this fact go as follows (see Figure 8 for an illustration):

- According to the semantics of AMASS, ρ contains at most k_t tasks, that is, $n \le k_t$.
- Suppose $S_i = [(B_{i,1}, \Theta_{i,1}), \cdots, (B_{i,l_i}, \Theta_{i,l_i})]$, and for each $i \in [n]$ and $j \in [l_i]$, $\Theta_{i,j} = (\upsilon_{i,j}, \eta_{i,j}, \iota_{i,j})$. Then for each $i \in [n]$, $l_i \le \hbar$, moreover, for each $j \in [l_i]$, $\upsilon_{i,j}$ contains at most k_c fragment containers and the height of each of them is no more than $\hbar$, and $\iota_{i,j}$ is a function from X to $\{0\} \cup [k_c\hbar]$ (since each fragment contains at most $\hbar$ instances in one fragment container and there are at most k_c fragment containers).
- Furthermore, $\eta_{i,j}$ contains at most $\hbar$ transactions, each of which has at most $k_a(\hbar + 1)$ concretized actions. To see this, note that $\eta_{i,j}$ is obtained by concretizing at most k_a actions in some transition rule of $\mathcal{M}$ and each such action is concretized into at most $\hbar + 1$ actions (because the heights of the history contents of fragment stacks are no more than $\hbar$). Note that each action $\beta(F', i', n')$ in the transactions of $\eta_{i,j}$ is encoded by a word of length 4.

More specifically, we have that

- each fragment container is encoded by a word of length $N_{ctn} = k_c\hbar$,
- each transaction is encoded by a word of length $N_{tra} = 4k_a(\hbar + 1)$,
- each transaction stack is encoded by a word of length $N_{trs} = \hbar N_{tra}$,
- each assignment function is encoded by a word of length $N_{asn} = 2k_i$, i.e. a word of the form $x_1 n_1 \cdots x_{k_i} n_{k_i}$ (where $x_j \in X$ and $n_j \in \{0\} \cup [k_c\hbar]$),
- each activity is encoded by a word of length $N_{act} = 1 + N_{ctn} + N_{trs} + N_{asn}$, and
- each task is encoded by a word of length $N_{tsk} = 2 + \hbar N_{act}$.

Note that in the encoding, the top activity of the top task is in the first position.

We can then use $k_t N_{tsk}$ variables $X_1, \cdots, X_{k_t N_{tsk}}$ ranging over $\Sigma_{\mathcal{M}}$ to encode a task stack. Evidently, each symbol in $\Sigma_{\mathcal{M}}$ can be encoded as a bit vector of length $\log_2 |\Sigma_{\mathcal{M}}|$. For each $\sigma \in \Sigma_{\mathcal{M}}$, assume $\mathrm{enc}(\sigma)$ is the binary encoding of σ. Each configuration can be encoded by $k_t N_{tsk} \log_2 |\Sigma_{\mathcal{M}}|$ Boolean variables. Therefore, $\mathcal{A}_{\mathcal{M}}$ uses $k_t N_{tsk} \log_2 |\Sigma_{\mathcal{M}}|$ state variables to encode the configurations in $\mathrm{Conf}_{\mathcal{M},c_t,\hbar}$. Moreover, $\mathcal{A}_{\mathcal{M}}$ contains the input variables to encode the transition rules of $\mathcal{M}$. Equipped with these state variables and input variables, we then encode the semantics of the transition rules of $\mathcal{M}$ into the transitions of $\mathcal{A}_{\mathcal{M}}$.

7 VALIDATION OF THE FORMAL SEMANTICS

The goal of this section is to validate the formal semantics of the AMASS models defined in Section 4. To avoid tediousness, we focus on the two sub-models AMASS_{ACT} and AMASS_{FRG} of AMASS and Android 13.0. We validate the formal semantics from two different perspectives.

- At first, we audit the Java source code of Android operating system to extract the control flows of starting an activity and executing a fragment transaction respectively. We confirm that the formal semantics of AMASS models defined in Section 4 indeed echo the extracted control flows. See Section 7.1 for more about the code audit.
- Moreover, we select 21 Android apps to empirically validate the conformance of the formal semantics of AMASS models to the actual behaviors of these apps in the Android operating system. The details of the empirical validation are given in Section 7.2.

7.1 Validation of the formal semantics by auditing the source code of Android OS

To validate the formal semantics of AMASS_{ACT} (resp. AMASS_{FRG}), we audit the source code of Android OS. Specifically, we trace the control flow of the Java source code[8] starting from the procedure startActivity() (resp. commit()). To increase the quality of the auditing process, we audit the source code in two phases. In the first phase, the three authors audit the relevant parts of the source code of Android OS separately. Then in the second phase, the three authors have a joint discussion and achieve a consensus on the understanding of the source code.

We first consider AMASS_{ACT}, then AMASS_{FRG}.

7.1.1 Auditing the source code of AMASS_{ACT}. Figure 9 shows the control flow starting from the procedure startactivity() in the Activity class, where the vertices represent the procedures in classes and edges represent the procedure calls.

- Starting from startactivity(), startActivityForResult() is called, then execStartActivity() is called, and so on, until the procedure startActivityInner() is reached, where the core control logic for starting an activity is implemented.
- In the procedure startActivityInner(), the following statements are executed sequentially: the procedure computeLaunchingTaskFlags(), a conditional statement where getReusableTask() and computeTargetTask() are in the two branches respectively, the procedure recycleTask(), and another conditional statement where setNewTask() and addOrReparentStartingActivity() are in the two branches respectively.
- The procedure getReusableTask() calls findTask(), in which the procedure process() is called, and process() calls the procedure forAllLeafTasks(), where the procedure test() is called.
- The procedure recycleTask() calls the procedure complyActivityFlags() and setNewTask() calls reuseOrCreateTask() respectively.

We audit the source code of the procedures called directly or indirectly by startActivityInner() and confirm that the semantics of AMASS_{ACT} is consistent with its actual implementation in Android OS. In particular, the task allocation

[8] available at https://android.googlesource.com/platform/frameworks/base/.

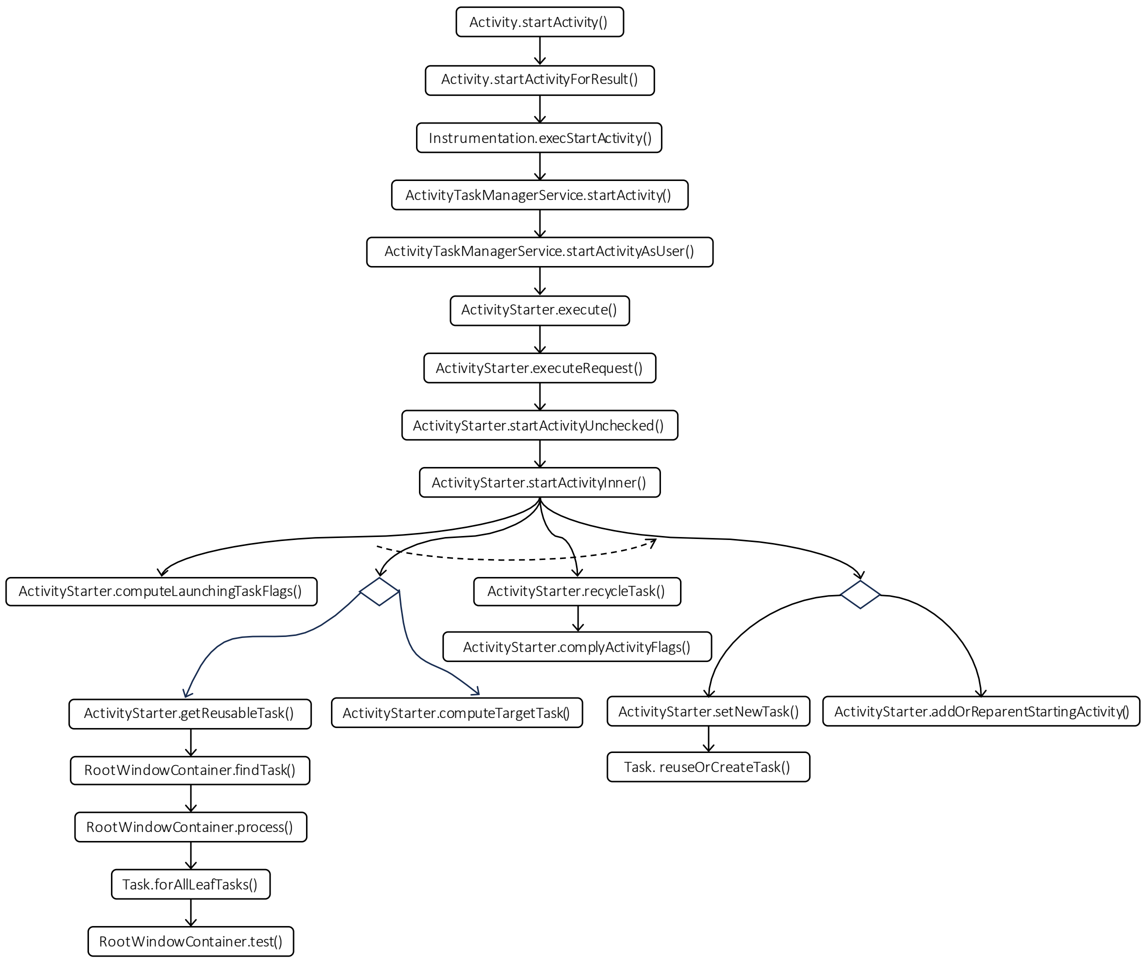

Fig. 9. Call graph for starting an activity

mechanism and the intent flags in the semantics conform to the source code in Android OS. The details of the auditing are relegated to Appendix E.1.

7.1.2 Auditing the source code for AMASS$_{FRG}$. The control flows of committing and popping a fragment transaction are illustrated in Figure 10.

- We start with committing a fragment transaction first. From Figure 10, to commit a fragment transaction, commit() is called. Then it calls commitInternal(), which calls enqueueAction() and so on until executeOpsTogether() is called, where the core control logic for committing a fragment transaction is implemented. The procedure executeOpsTogether() calls expandOps() first, then executeOps(). Finally, executeOps() calls BackStackRecord.executeOps().

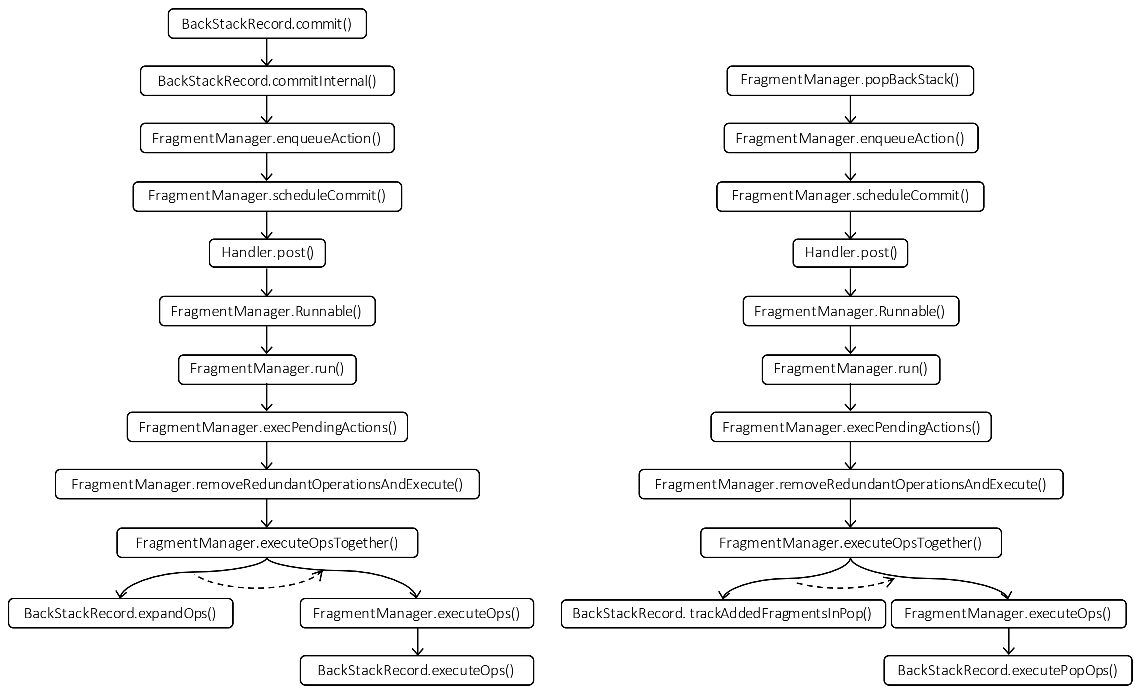

Fig. 10. Call graphs for committing and popping a fragment transaction

- The call graph for popping a fragment transaction is similar. From Figure 10, to pop a fragment transaction, popBackStack() is called. Then it calls commitInternal(), which calls enqueueAction() and so on until executeOpsTogether() is called. The procedure executeOpsTogether() calls trackAddedFragmentsInPop() first, then executeOps(). Finally, executeOps() calls BackStackRecord.executePopOps().

We audit the source codes of the procedure executeOpsTogether() as well as those called by it directly or indirectly and confirm that

- the way of dealing with REP actions in the semantics of AMASS$_{FRG}$ in Section 4.1.2 is consistent with the source code of expandOps(),
- the execution of fragment actions as defined in the semantics of AMASS$_{FRG}$ in Section 4.1.2 is consistent with the source code of BackStackRecord.executeOps(),
- the way of revoking fragment transactions in the semantics of AMASS$_{FRG}$ in Section 4.1.2 is consistent with the source code of trackAddedFragmentsInPop().

The details of the auditing are relegated to Appendix E.2.

7.2 Empirical validation of the formal semantics

The goal of this section is to validate the formal semantics empirically by comparing the actual behaviors of Android apps with their expected behaviors according to the formal semantics of the AMASS models that are extracted from them. To ease the empirical validation, we try to automate the process and propose a partially automated method for semantics validation in the sequel, by utilizing the nuXmv model checker, the Android UIAutomator testing framework, and the ADB tool.

Partially automated method for semantics validation. Let X be an Android app and $\mathcal{M}$ be an AMASS$_{ACT}$ (resp. AMASS$_{FRG}$) model that is extracted from X and $\mathcal{A}$ be the nuXmv FSM model that encodes the configuration reachability problem of $\mathcal{M}$, where c_t, the bound on the number of tasks of the same affinity, and $\hbar$, the bound on the height of the stacks, are set to be 2 and 6 respectively. Moreover, k_a, the number of actions in one fragment transaction, is set to be 2.

The partially automated method to validate the semantics goes as follows. For a transition rule τ in $\mathcal{M}$,

(1) we use nuXmv to search for a path $\pi = \pi_1 \cdots \pi_m$ in $\mathcal{A}$, where an accepting condition corresponding to τ is set, in order to reach a configuration where τ can be applied (in other words, τ is enabled by the configuration),
(2) we generate from the app X and each transition π_i with $i \in [m]$, a sequence of click events in X, say e_i,
(3) we use UIAutomator to execute the click events in $e_1, \cdots, e_m$ one by one (the task stack is updated), moreover, after the execution of all the click events, we use ADB to extract the resulting configuration, that is, the contents of the resulting task stack, say ρ,
(4) we generate the sequence of click events in X, say e, to fulfill the application of the transition rule τ,
(5) we use UIAutomator again to execute the click events in e one by one, and use ADB to obtain the resulting configuration ρ',
(6) if $\rho' \neq \tau(\rho)$, then report the semantic inconsistency, where $\tau(\rho)$ denotes the expected configuration obtained by applying τ on ρ according to the formal semantics.

Note that the generation of sequences of click events for transition rules (namely, the second and fourth step in the aforementioned procedure) is *not* automated. For a transition rule of the form $\tau = A \xrightarrow{\alpha(\phi)} B$, we can click the widgets in the activity A in order to find a sequence of click events so that the activity B can be launched. Nevertheless, just from the sequence of click events and without the manual inspection of the source code, it is difficult, if not impossible, to figure out the intent flags associated with the action of starting the activity B from the activity A. As a result, it is hard to determine that a sequence of click events corresponds to the transition rule τ, even though we do know that the activity B is started from A.

Since generating sequences of click events for transitions is a manual process, it is hard to apply the aforementioned partially automated method for semantics validation to a large pool of Android apps from F-Droid or Google Play. As a result, in the sequel, we only select a small number of (more precisely, 20) apps from F-Droid to validate the formal semantics.

- At first, we select 10 apps to cover as many launch modes and intent flags as possible for validating the semantics of AMASS$_{ACT}$.
- Then, we select 10 apps to cover as many types of fragment transactions as possible for validating the semantics of AMASS$_{FRG}$.

We generate the sequences of click events from the transition rules by manually inspecting the source codes of these 20 apps. Evidently, these 20 apps are insufficient to cover the vast number of cases in the definition of the semantics of AMASS$_{ACT}$ and AMASS$_{FRG}$, since there are exponentially many different combinations of intent flags. Therefore, in addition to these apps, we design a special Android app ValApp[9], aiming at covering all the cases in the definition of the semantics of AMASS$_{ACT}$ and AMASS$_{FRG}$ models. We mainly rely on ValApp to enumerate the different combinations

[9] available at https://github.com/Jinlong-He/ValApp/tree/main

of intent flags and carry out semantics validation for them. Note that this way is largely credible since, after the options (e.g. launch modes and intent flags) are chosen, the rest of the validation process is the same as real-world Android apps, that is, the startActivity() or commit() procedure is executed in the Android OS and the results are collected and compared.

ValApp: A special Android app for the semantics validation. A snapshot of ValApp is shown in Figure 11. ValApp includes 8 activities, corresponding to the 8 (launch mode, task affinity) pairs from $\{\mathrm{STD}, \mathrm{STP}, \mathrm{STK}, \mathrm{SIT}\} \times \{1, 2\}$. Moreover, ValApp provides means to choose the launch modes and intent flags, as well as "start" and "finishStart", so that transition rules can be generated for all their combinations. Furthermore, each activity of ValApp includes two fragments and two fragment containers. It also provides two variables for fragment identifiers and allows arbitrary combinations of actions to form fragment transactions. Finally, ValApp allows pushing arbitrarily many transactions into the fragment transaction stack. From the design, we can see that ValApp is capable of comprehensively accounting for all factors that may affect the semantics of AMASS$_{ACT}$ and AMASS$_{FRG}$ models. Moreover, ValApp is *universal* in the sense that a user can interact with ValApp to choose the transition rules and generate a desired AMASS model.

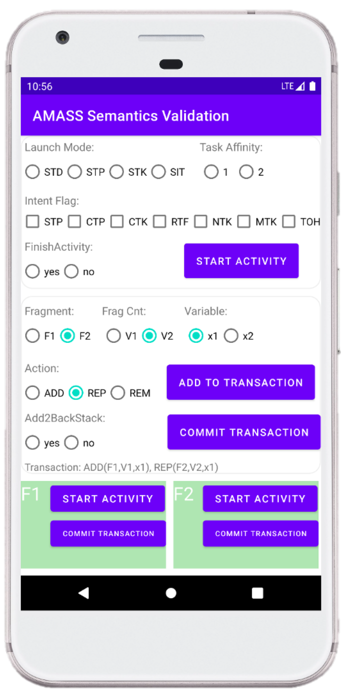

Fig. 11. ValApp: An Android app designed for validating the semantics of AMASS

Let us focus on Android 13.0 in this section and validate the semantics of AMASS$_{ACT}$ and AMASS$_{FRG}$ models under this version. The verification of the formal semantics of AMASS$_{ACT}$ and AMASS$_{FRG}$ under the other versions is similar and omitted.

7.2.1 *Validation of the formal semantics of AMASS$_{ACT}$.*

To valid the semantics, we consider the transition rules of the form $\tau = A \xrightarrow{\alpha(\phi)} B$, and generate one configuration for each combination of the launch modes of A and B, the values of α, the intent flags, and the constraints on the configuration before applying the transition, e.g. $\mathrm{GetRealTsk}(\rho, B) = *$, $\mathrm{GetTsk}(\rho, B) = S_1$, and $\mathrm{TopAct}(\rho) = B$. In total, there are $901,120$ different combinations to be considered and we would like to generate configurations for all of them.

We utilize the 10 apps selected from F-Droid as well as ValApp to generate the configurations. The 10 F-Droid apps are selected to cover as many launch modes and intent flags as possible. The sizes of these apps, the number of activities and transition rules of the AMASS$_{ACT}$ models constructed out of these apps (i.e. $|\mathrm{Act}|$ and $|\Delta|$), as well as the number of generated configurations can be found in Table 5. In the end, we generate $6,231$ configurations out of the F-Droid apps and $901,120$ configurations out of ValApp. Then we use these configurations to validate the formal semantics. Through experiments, we discover that for every combination, the configuration obtained by applying the transition rule corresponding to the combination according to the formal semantics and the actual configuration returned by ADB are equal, thus the formal semantics of AMASS$_{ACT}$ for Android 13.0 are confirmed to be consistent with the actual behaviors of Android apps.

App (package name)	size (MB)	$\lvert\mathrm{Act}\rvert$	$\lvert\Delta\rvert$	# configurations
com.fsck.k9	3.58	21	76	892
org.andstatus.app	3.37	12	29	203
com.fsck.k9.material	4.49	17	53	686
org.videolan.vlc	16.16	16	36	567
com.github.kiliakin.yalpstore	1.43	14	119	1, 195
one.librem.mail	4.69	18	74	984
one.librem.tunnel	20.38	6	34	378
eu.siacs.conversations.legacy	11.12	17	82	993
org.fdroid.k9	3.12	20	70	893
info.guardianproject.otr.app.im	10.7	12	43	492
Total	-	-	-	6, 231
ValApp	-	8	131, 072	901, 120

Table 5. Validation of the formal semantics of AMASS$_{ACT}$ for Android 13.0

7.2.2 *Validation of the formal semantics of AMASS$_{FRG}$.*

To valid the semantics, we consider the transition rules of the form $\tau = A \xrightarrow{\mu} T$ or $F \xrightarrow{\mu} T$. Note that according to the definition of semantics of AMASS$_{FRG}$ in Section 4.1.2, the only requirement for the enablement of $A \xrightarrow{\mu} T$ or $F \xrightarrow{\mu} T$ is that the top activity or fragment is A or F. This requirement does not constrain the configurations very much. To validate the semantics of AMASS$_{FRG}$, we fix the values of the following parameters and generate configurations as well as transition rules with these values.

- there is only one activity, say A_0,
- the number of containers associated with A_0 is 1,
- the maximum number of transactions in the transaction stack is 1,
- the maximum number of fragment actions in a transaction is 2,
- the identifiers in (concretized) fragment actions in a transaction are from the set $\{1, 2\}$.

Note that the maximum number of fragments in a container is bounded by $\hbar = 6$ (i.e. the bound on the height of the stacks). In total, there are 4032 different configurations to be considered. We generate the 4032 configurations,

App (package name)	size(MB)	$\|Frg\|$	$\|\Delta\|$	**average #actions/transaction**	**#(config, trans)**
org.fox.ttrss	1.15	9	18	1.2	26,405
exa.lnx.a	2.88	10	18	1.0	21,403
org.wordpress.android	4.31	7	19	1.1	7,028
de.geeksfactory.opacclient	4.06	4	23	1.0	1,919
se.oandell.riksdagen	2.66	6	23	1.5	31,012
com.secuso.privacyFriendlyCodeScanner	1.62	6	24	1.0	28,425
com.igisw.openmoneybox	9.04	9	28	1.9	38,543
xyz.hisname.fireflyiii	4.82	16	33	1.1	64,160
org.anhonesteffort.flock	4.22	8	41	1.0	51,507
dulleh.akhyou.fdroid	4.17	4	126	1.8	84,602
Total	-	-	-	-	355,004
ValApp	-	2	577	2.0	1,745,856

Table 6. Validation of the semantics of AMASS_{FRG} models

construct AMASS_{FRG} models out of these apps, apply the transition rules in the models to the generated configurations to validate the semantics.

We utilize the 10 apps selected from F-Droid as well as ValApp to validate the semantics. The 10 F-Droid apps are selected to include as many fragments and fragment transactions as possible. Moreover, when constructing AMASS_{FRG} models out of the F-Droid apps, we restrict our attention to an activity whose number of fragments is the greatest in the app. For ValApp, since the number of actions in a transaction can be unbounded, we restrict our attention to the transition rules where the number of actions in a transaction is at most 2. The sizes of these apps, the number of activities and transition rules of the AMASS_{FRG} models constructed out of these apps (i.e. $|Frg|$ and $|\Delta|$), and the average number of actions per transaction, as well as the number of generated (configuration, transition rule) pairs can be found in Table 5.

In the end, we generate 4,032 configurations for each of the F-Droid apps and ValApp. We also generate 577 transition rules for ValApp. Therefore, the total number of (configuration, transition rule) pairs for F-Droid apps are 355,004, while the number for ValApp is 1,745,856. Then for all these pairs, we apply the transition rules on the configurations to validate the formal semantics. Through experiments, we discover that for each configuration and each transition rule, the configuration obtained by applying the transition rule according to the formal semantics and the actual configuration returned by ADB are equal, thus the formal semantics of AMASS_{FRG} for Android 13.0 are confirmed to be consistent with the actual behaviors of Android apps.

8 STATIC ANALYSIS OF AMASS MODELS

In this section, we consider the static analysis of AMASS models. We focus on two vulnerabilities which might occur in Android apps, i.e., *task unboundedness vulnerability* and *fragment-container unboundedness vulnerability*. The former refers to the case that in the execution of the app, one might be in a configuration the height of which is unbounded. The latter one is similar, where the height of a fragment container of some activity of the configuration is unbounded. As we will see later, these vulnerabilities would cause abnormal behavior of apps. The stack analysis aims to detect the apps who are potentially exposed to the vulnerabilities. While the problem is generally unsolvable in theory, we appeal to sensible relaxations, as illustrated in the following two subsections.

8.1 Task unboundedness

To detect the task unboundedness vulnerability, we consider k-*task unbounded* where k is a given natural number. An AMASS $\mathcal{M}$ is k-*task unbounded* if, for every $n \in \mathbb{N}$, there are a configuration ρ of $\mathcal{M}$ and a task in ρ such that $\epsilon \xrightarrow{\mathcal{M}} \rho$, the height of the task is at least n, and the path from ϵ to ρ involves the interplay with at most k other tasks.

For instance, we assume that in $\mathcal{M}$ the following two conditions are satisfied:

(i) the main activity A_0 satisfies $\mathsf{Aft}(A_0) = 1$, the main task is the top task, and the top activity of the main task is $A \neq A_0$ with $\mathsf{Lmd}(A) = \mathsf{STD}$ and $\mathsf{Aft}(A) = 1$,

(ii) $\mathcal{M}$ contains two transition rules $A \xrightarrow{\mathsf{start}(\phi_1)} B$ and $B \xrightarrow{\mathsf{start}(\phi_2)} A$ such that $\mathsf{Lmd}(B) = \mathsf{STK}$, $\mathsf{Aft}(B) = 2$, $\phi_1 \models \mathsf{CTK}$, $\phi_2 \models \mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{MTK} \wedge \neg\mathsf{STP} \wedge \neg\mathsf{RTF} \wedge \neg\mathsf{CTP} \wedge \neg\mathsf{CTK}$.

Then $\mathcal{M}$ is 1-task unbounded since the number of A instances in the main task can go unbounded by executing the two transition rules repeatedly, during which the main task interacts with another task where B is the only activity.

We hypothesize that, most task unbounded vulnerabilities only involve a small number of tasks (normally, $k \leq 2$). which will be justified by experiments in Section 9.3. As a result, we introduce algorithms to solve the k-task unboundedness problem. We will start with $k = 0$ and $k = 1$, then consider the more general case $k \geq 2$.

We will focus on the following typical situation: For any pair of distinct tasks, if none of their real activities has the SIT (SingleInstance) launch mode, then their affinities should be different. Note that the constraint in this situation is satisfied, if in all the transitions of $\mathcal{M}$, MTK is set to be false, namely, $\phi \models \neg\mathsf{MTK}$. This assumption is empirically justified by the fact that in 6,388 open-source F-Droid and commercial Google-play apps used for experiments (see Section 9), only less than 3% percent of them contain occurrences of the MTK flag.

We first introduce some notations. We use $\mathsf{Act}_{\mathsf{real}}$ to denote the set of activities in $\mathcal{M}$ that may occur as a real activity of tasks. More precisely, $\mathsf{Act}_{\mathsf{real}}$ is the set of activities $A \in \mathsf{Act}$ such that one of the following conditions holds: 1) $\mathsf{Lmd}(A) = \mathsf{SIT}$, 2) $\mathsf{Lmd}(A) = \mathsf{STK}$, 3) $\mathsf{Lmd}(A) = \mathsf{STD}$ or STP, and A occurs in some transition $B \xrightarrow{\alpha(\phi)} A$ such that $\mathsf{Lmd}(B) = \mathsf{SIT}$ or $\phi \models \mathsf{NTK} \vee \mathsf{NDM}$.

For each activity $A \in \mathsf{Act}_{\mathsf{real}}$ such that $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, let $\mathsf{Reach}(\Delta, A)$ denote the least subset $\Theta \subseteq \Delta$ satisfying that $B \xrightarrow{\alpha(\phi)} C \in \Theta$ (where $\alpha = \mathsf{start}$ or $\mathsf{finishStart}$) whenever the following two constraints are satisfied:

- $B = A$ or there exists a transition $A' \xrightarrow{\alpha'(\phi')} B \in \Theta$ (where $\alpha' = \mathsf{start}$ or $\mathsf{finishStart}$),
- $\mathsf{Lmd}(C) \neq \mathsf{SIT}$, and if $\mathsf{Lmd}(C) = \mathsf{STK}$ or $\phi \models \mathsf{NTK} \vee \mathsf{NDM}$, then $\mathsf{Aft}(C) = \mathsf{Aft}(A)$.

Intuitively, $\mathsf{Reach}(\Delta, A)$ comprises all the transition rules that can be applied and once applied would retain an A-task as the top task. By abusing the notation slightly, $\mathsf{Reach}(\Delta, A)$ also denotes the graph whose edge set is $\mathsf{Reach}(\Delta, A)$.

$\mathsf{Reach}(\Delta, A)$ can be generalized to the case that $A \in \mathsf{Act}_{\mathsf{real}}$ and $\mathsf{Lmd}(A) = \mathsf{SIT}$, where $\mathsf{Reach}(\Delta, A)$ is regarded as the graph that contains a single node A without edges.

Case $k = 0$.

Now we will propose a procedure to check k-task unboundedness for $k = 0$. The underpinning idea is to search, for each $A \in \mathsf{Act}_{\mathsf{real}}$, a *witness cycle*, i.e., a sequence of transitions from $\mathsf{Reach}(\Delta, A)$, the execution of which would force the task to grow indefinitely.

Formally, a witness cycle is a simple cycle in the graph $\mathsf{Reach}(\Delta, A)$ of the form

$$C = A_1 \xrightarrow{\alpha_1(\phi_1)} A_2 \cdots A_{n-1} \xrightarrow{\alpha_{n-1}(\phi_{n-1})} A_n$$

where $n \geq 2$ and α_i = start or finishStart for each $i \in [n]$ satisfying the following two constraints:

[Non-clearing] The content of an A-task is *not* cleared when C is executed. Namely, for each $i \in [n-1]$, $\phi_i \models \neg\mathsf{CTP} \wedge \neg\mathsf{NDM}$, moreover, either $\phi_i \models \neg\mathsf{CTK}$, or $\phi_i \models \neg\mathsf{NTK}$ and $\mathsf{Lmd}(A_{i+1}) \neq \mathsf{STK}$ (intuitively, this means that CTK is not enabled, cf. Figure 19).

[Height-increasing] The height of the task content is increasing after C is executed. Namely, it is required that $\sum_{i\in[n-1]} weight_C(\tau_i) > 0$, where for each $i \in [n-1]$, $\tau_i = A_i \xrightarrow{\alpha_i(\phi_i)} A_{i+1}$ and $weight_C(\tau_i)$ specifies the modification of the height of the A-task by executing τ_i and is defined as follows.

- If α_i = start, then
  - if $\phi_i \models \mathsf{RTF}$, then $weight_C(\tau_i) = 0$,
  - if $\phi_i \models \neg\mathsf{RTF}$, $A_i = A_{i+1}$, and either $\phi_i \models \mathsf{STP}$ or $\mathsf{Lmd}(A_{i+1}) = \mathsf{STP}$, then $weight_C(\tau_i) = 0$,
  - otherwise, $weight_C(\tau_i) = 1$.
- If α_i = finishStart, then
  - if $\phi_i \models \mathsf{RTF}$, then $weight_C(\tau_i) = -1$,
  - if $\phi_i \models \neg\mathsf{RTF}$, $A_i = A_{i+1}$, and either $\phi_i \models \mathsf{STP}$ or $\mathsf{Lmd}(A_{i+1}) = \mathsf{STP}$, then $weight_C(\tau_i) = -1$,
  - otherwise, $weight_C(\tau_i) = 0$.

If a witness cycle exists for some $A \in \mathsf{Act}_{\mathsf{real}}$, the algorithm returns "task unbounded"; otherwise, if Δ is a directed acyclic graph, then the algorithm returns "task bounded"; otherwise, the procedure reports "unknown".

The more general cases for $k \geq 1$ are much more technical and involved. We introduction the concept of "virtual transitions" for tasks to capture the situation that the content of a task can be indirectly modified by first jumping off the task and returning to the task later on. When this happens, the procedure adds virtual transitions for each task before checking the existence of witness cycles.

Case $k = 1$.

Let $A \in \mathsf{Act}_{\mathsf{real}}$. In this case, we check whether the height of some A-task is unbounded, but involving another task. The main difficulty of this case, in a contrast to $k = 0$, is that, when the system evolves, the A-task may give away as the top task, i.e., the other task may become the top task, but then gives back to the A-task later on. Even worse, the content of the A-task may have be changed during the round of switch over, and there may be several rounds during which the height of the A-task grows. To accommodate such a complex analysis, we will utilise a concept of *virtual transitions* to summarize the changes of the content of the A-task for each round of top task switching.

We introduce some notations first. Let $A, B \in \mathsf{Act}_{\mathsf{real}}$ such that $\mathsf{Lmd}(A) \neq \mathsf{SIT}$ and A, B represent different tasks, specifically, one of the following conditions holds:

(1) $\mathsf{Lmd}(A) = \mathsf{Lmd}(B) = \mathsf{SIT}$ and $A \neq B$,
(2) $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\mathsf{Lmd}(B) \neq \mathsf{SIT}$,
(3) $\mathsf{Lmd}(A) \neq \mathsf{SIT}$ and $\mathsf{Lmd}(B) = \mathsf{SIT}$,
(4) $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, $\mathsf{Lmd}(B) \neq \mathsf{SIT}$, and $\mathsf{Aft}(A) \neq \mathsf{Aft}(B)$.

Let $G_A = (V, E)$ be an edge-labeled graph, where the edge labels are of the form $\alpha(\phi)$ with $\alpha \in \{\mathsf{start}, \mathsf{finishStart}\}$ and $\phi \in \mathscr{B}(\mathcal{F})$. Intuitively, G_A is used to capture the set of transitions that can be applied to update the content of an A-task.

A *task-switching transition from G_A to a B-task* is a transition $A' \xrightarrow{\alpha'(\phi')} B' \in \Delta$ such that $A' \in V$, and one of the following conditions holds,

- $\mathsf{Lmd}(B') = \mathsf{SIT}$ and $B' = B$,
- $\mathsf{Lmd}(B') = \mathsf{STK}$, moreover, $\mathsf{Aft}(B') = \mathsf{Aft}(B)$ and $\mathsf{Lmd}(B) \neq \mathsf{SIT}$,
- $\mathsf{Lmd}(B') = \mathsf{STD}$ or STP, and $\phi' \models \mathsf{NTK} \vee \mathsf{NDM}$, moreover, $\mathsf{Aft}(B') = \mathsf{Aft}(B)$ and $\mathsf{Lmd}(B) \neq \mathsf{SIT}$.

Intuitively, a task-switching transition from G_A to a B-task means the task-switching in the situation that A-task and B-task already exist. Note that in this situation, if $\mathsf{Lmd}(B) \neq \mathsf{SIT}$ and $\mathsf{Lmd}(B') \neq \mathsf{SIT}$, then there does not exist a B'-task, since there do not exist two non-SIT-tasks whose affinities are $\mathsf{Aft}(B)$.

A transition $A' \xrightarrow{\alpha'(\phi')} A''$ is a *virtual transition of G_A w.r.t. B* if there is a task-switching transition $A' \xrightarrow{\alpha(\phi)} B'$ from G_A to a B-task and a task-switching transition $B'' \xrightarrow{\alpha'(\phi')} A''$ from $\mathsf{Reach}(\Delta, B')$ to an A-task. Note that the virtual transition inherits the intent-flag constraint of $B'' \xrightarrow{\alpha'(\phi')} A''$.

The *completion of* $\mathsf{Reach}(\Delta, A)$ *by the virtual transitions w.r.t. B*, denoted by $\mathsf{Comp}_B(\mathsf{Reach}(\Delta, A))$, is defined as the graph computed by the following three-step algorithm.

(1) Let $G_0 := \mathsf{Reach}(\Delta, A)$ and $i := 0$.
(2) Iterate the following procedure until $G_{i+1} = G_i$: Let G_{i+1} be the graph obtained from G_i by adding all the virtual transitions of G_i w.r.t. B. Let $i := i + 1$.
(3) Let $\mathsf{Comp}_B(\mathsf{Reach}(\Delta, A)) := G_i$.

Finally, we present the algorithm to decide the 1-task unboundedness of $\mathcal{M}$.

Algorithm to decide the 1-task unboundedness of $\mathcal{M}$

Check for each $A, B \in \mathsf{Act}_{\mathsf{real}}$ such that $\mathsf{Lmd}(A) \neq \mathsf{SIT}$ and A, B *represent different tasks*, whether there exists a witness cycle in $\mathsf{Comp}_B(\mathsf{Reach}(\Delta, A))$.
If affirmative, then the algorithm returns "task unbounded".
Otherwise, if Δ is a directed acyclic graph, then the algorithm returns "task bounded".
Otherwise, the algorithm returns "unknown".

Case $k \geq 2$.

We first extend the concept of representing different tasks from two to multiple tasks. For $A \in \mathsf{Act}_{\mathsf{real}}$ and $\mathbb{A} = \{A_1, \cdots, A_k\} \subseteq \mathsf{Act}_{\mathsf{real}}$, we say that $A, A_1, \cdots, A_k$ represent different tasks if they are mutually distinct, in addition, for each pair of them, say A' and A'', they represent different tasks.

Let $A \in \mathsf{Act}_{\mathsf{real}}$ and $\mathbb{A} = \{A_1, \cdots, A_k\} \subseteq \mathsf{Act}_{\mathsf{real}}$ such that $A, A_1, \cdots, A_k$ represent different tasks in the sequel.

Let $G = (V, E)$ be an edge-labeled graph, where the edge labels are of the form $\alpha(\phi)$ with $\alpha \in \{\mathsf{start}, \mathsf{finishStart}\}$ and $\phi \in \mathscr{B}(\mathcal{F})$, with the intention of capturing the set of transitions that can be applied to update the content of an A-task. Note that if $\mathsf{Lmd}(A) = \mathsf{SIT}$, then G is the graph that contains a single node A and no edges. A *task-switching transition from G to $\mathbb{A}$-tasks* is a transition $A' \xrightarrow{\alpha'(\phi')} B' \in \Delta$ such that $A' \in V$ and one of the following constraints holds,

- $\mathsf{Lmd}(B') = \mathsf{SIT}$ and $B' \in \mathbb{A}$,
- $\mathsf{Lmd}(B') = \mathsf{STK}$ and $\mathsf{Aft}(B') \in \mathsf{Aft}(\mathbb{A} \setminus \mathsf{Act}_{\mathsf{SIT}})$,
- $\mathsf{Lmd}(B') = \mathsf{STD}$ or STP, and $\phi' \models \mathsf{NTK} \vee \mathsf{NDM}$, moreover, $\mathsf{Aft}(B') \in \mathsf{Aft}(\mathbb{A} \setminus \mathsf{Act}_{\mathsf{SIT}})$.

Intuitively, a task-switching transition from G to $\mathbb{A}$-tasks means the task-switching in the situation that A-task, and A_1-task, $\cdots$, A_k-task already exist.

For $B' \in \mathsf{Act}$ such that either $\mathsf{Lmd}(B') = \mathsf{SIT}$ and $B' \in \mathbb{A}$, or $\mathsf{Lmd}(B') \neq \mathsf{SIT}$ and $\mathsf{Aft}(B') \in \mathsf{Aft}(\mathbb{A} \setminus \mathsf{Act}_{\mathsf{SIT}})$, we use $\mathsf{Reach}_{\mathbb{A}}(\Delta, B')$ to denote the least set of transitions $\Theta \subseteq \Delta$ satisfying the following constraints:

- $\mathsf{Reach}(\Delta, B') \subseteq \Theta$,
- for each transition $C \xrightarrow{\alpha(\phi)} D \in \Delta$ satisfying the following two conditions, we have $C \xrightarrow{\alpha(\phi)} D \in \Theta$:
  (1) C is either B' or the target node of some transition in Θ,
  (2) if $\mathsf{Lmd}(D) = \mathsf{SIT}$, then $D \in \mathbb{A}$,
  if $\mathsf{Lmd}(D) = \mathsf{STK}$, then $\mathsf{Aft}(D) \in \mathsf{Aft}(\mathbb{A} \setminus \mathsf{Act}_{\mathsf{SIT}})$,
  if $\mathsf{Lmd}(D) = \mathsf{STD}$ or STP, and $\phi \models \mathsf{NTK} \vee \mathsf{NDM}$, then $\mathsf{Aft}(D) \in \mathsf{Aft}(\mathbb{A} \setminus \mathsf{Act}_{\mathsf{SIT}})$.

Intuitively, $\mathsf{Reach}_{\mathbb{A}}(\Delta, B')$ is the set of transitions that are reachable from B' that can be used to update the contents of the tasks whose real activities are from $\mathbb{A}$.

Suppose additionally $\mathsf{Lmd}(A) \neq \mathsf{SIT}$ holds. For a graph $G = (V, E)$ capturing the set of transitions that can be applied to update the content of an A-task, a *virtual transition of* G *w.r.t.* $\mathbb{A}$ is some $A' \xrightarrow{\alpha'(\phi')} A''$ such that there is a task-switching transition $A' \xrightarrow{\alpha(\phi)} B'$ from G to $\mathbb{A}$-tasks and a task-switching transition $B'' \xrightarrow{\alpha'(\phi')} A''$ from $\mathsf{Reach}_{\mathbb{A}}(\Delta, B')$ to an A-task. We define the *completion of* $\mathsf{Reach}(\Delta, A)$ *by the virtual transitions w.r.t.* $\mathbb{A}$, denoted by $\mathsf{Comp}_{\mathbb{A}}(\mathsf{Reach}(\Delta, A))$, as the graph computed by the following three-step algorithm.

(1) Let $G_0 := \mathsf{Reach}(\Delta, A)$ and $i := 0$.
(2) Iterate the following procedure until $G_{i+1} = G_i$: Let G_{i+1} be the graph obtained from G_i by adding all the virtual transitions of G_i w.r.t. $\mathbb{A}$. Let $i := i + 1$.
(3) Let $\mathsf{Comp}_{\mathbb{A}}(\mathsf{Reach}(\Delta, A)) := G_i$.

We are ready to present the procedure to solve the k-task boundedness problem for $k \geq 2$.

Algorithm to solve the k-task boundedness problem for $k \geq 2$

For each $A \in \mathsf{Act}_{\mathsf{real}}$ and $\mathbb{A} = \{A_1, \cdots, A_k\} \subseteq \mathsf{Act}_{\mathsf{real}}$ such that $A, A_1, \cdots, A_k$ represent different tasks and $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, check whether there exists a witnessing cycle in the graph $\mathsf{Comp}_{\mathbb{A}}(\mathsf{Reach}(\Delta, A))$.
If the answer is yes for any such a pair A and $\mathbb{A}$, then the procedure reports "task unbounded". Otherwise, if Δ is a directed acyclic graph, then the procedure reports "task bounded".
Otherwise, the procedure reports "unknown".

Generation of the witnessing transition sequence

As mentioned before, task unboundedness suggests a potential security vulnerability. As a result, when this is spotted, it is desirable to synthesize a concrete transition sequence so that the developers can, for instance, follow this sequence to test and improve their apps. It turns out that the synthesis can be reduced to the the configuration reachability problem of the FSM $\mathcal{A}_{\mathcal{M}}$, i.e., whether there is a configuration matching a prescribed formula is reachable.

Moreover, for a witness cycle for some $A \in \mathsf{Act}_{\mathsf{real}}$, $C = A_1 \xrightarrow{\alpha_1(\phi_1)} A_2 \cdots A_{n-1} \xrightarrow{\alpha_{n-1}(\phi_{n-1})} A_n$ When $k > 0$, some virtual transition, i.e., $A_i \xrightarrow{\alpha_i(\phi_i)} A_{i+1} \notin \Delta$, but it summarizes the changes of the content of the A-task for some round of top task switching. Hence the existence of a witness cycle for A is reduced to the existence of the activity sequence C' in the A-task, where C' is obtained by the following procedure:

(1) Let $C' := [A_1]$, $i := 1$,
(2) Iterate the following procedure until $i = n$,
  - if $\alpha_i = \mathsf{start}$,
    - if $A_i = A_{i+1}$ and $\phi_i \models \mathsf{STP} \vee \mathsf{RTF}$, then $C' := C'$,
    - otherwise, $C' := [A_{i+1}] \cdot C'$,
  - if $\alpha_i = \mathsf{finishStart}$, let $C' = [A_i] \cdot C''$, $C'' \in \mathsf{Act}^*$,
    - if $A_i = A_{i+1}$ and $\phi_i \models \mathsf{STP} \vee \mathsf{RTF}$, then $C' := C''$,
    - otherwise, $C' := [A_{i+1}] \cdot C''$.

Note that among the activities in C' the leftmost activity of C' is the topmost in the A-task.

From the encoding of task stacks in Section 6, we know that for each $i : 1 \leq i \leq k_t$,

> $X_{iN_{tsk}-1}$ corresponds to the second to the last position of the word encoding the ith task and its value is the real activity of the task.

Therefore, the existence of the activity sequence $C' = [A'_1, \ldots, A'_{n'}]$ (where $n' < \hbar$) in the A-task can be specified by the following formula,

$$X_{N_{tsk}-1} = A \ \wedge \bigvee_{0 \leq j \leq \hbar - n'} \ \bigwedge_{1 \leq k \leq n'} X_{(j+k-1)(1+N_{ctn}+N_{trs}+N_{asn})+1} = A'_k.$$

Therefore, the witnessing transition sequences can be generated by utilizing nuXmv to solve the resulting model checking problem, where $\hbar$ is set to 6.

8.2 Fragment container unboundedness problem

Recall that fragment containers are updated by the fragment transactions, that is, either all the actions in a transaction are executed or none of them are executed. Therefore, to detect the fragment container unboundedness vulnerability, it is necessary to reason at the level of fragment transactions rather than fragment actions. Similar to the previous case, the general idea is to find witness cycle of fragment transactions.

We start with introducing some notations to summarize the effects of fragment transactions on fragment containers. Let A be an activity with $\mathsf{Ctn}(A) = (i_1, \cdots, i_k)$ and

$$T = (\beta_1(F_1, i'_1, x_1), \cdots, \beta_k(F_r, i'_r, x_r))$$

be a fragment transaction such that $i'_j \in \{i_1, \cdots, i_k\}$ for each $j \in [r]$.

We define $U_{T,A,i_1}, \cdots, U_{T,A,i_k}$ as the subsequences of actions in T that are applied to the container $i_1, \cdots, i_k$ respectively. Specifically, $U_{T,A,i_1}, \cdots, U_{T,A,i_k}$ can be computed inductively as follows.

(1) Initially let $j = 1$ and $U_{T,A,i_1} = \epsilon, \ldots, U_{T,A,i_k} = \epsilon$.
(2) Iterate the following procedure until $j > r$,
  (a) update U_{T,A,i'_j} according to β_j,
    - if $\beta_j = \mathsf{REP}$, then let $U_{T,A,i'_j} = \mathsf{REP}(F_j)$,
    - otherwise, let $U_{T,A,i'_j} = U_{T,A,i'_j} \cdot \beta_j(F_j)$.
  (b) let $j = j + 1$.

Note in the computation of $U_{T,A,i_1}, \cdots, U_{T,A,i_k}$, the identifier variables are ignored.

Moreover, for $j \in [k]$, we define the *weight* of T with respect to the container i_j, denoted by $\mathsf{weight}_{T,A,i_j}$, which describes the update of T on the height of container i_j, as follows. Let $U_{T,A,i_j} = (\beta'_1(F'_1), \cdots, \beta'_l(F'_l))$.

- If $\beta'_r(F'_r) = \mathsf{REP}$ for some $r \in [l]$ (actually in this case, $l = 1$), then $\mathsf{weight}_{T,A,i_j} = -\infty$,
- otherwise, $\mathsf{weight}_{T,A,i_j} = \sum_{r \in [l]} w_r$, where for $r \in [l]$, $w_r = 1$ if $\beta'_r = \mathsf{ADD}$, and $w_r = -1$ if $\beta'_r = \mathsf{REM}$.

Moreover, let $\mathsf{weight}_{T,A} = (\mathsf{weight}_{T,A,i_1}, \cdots, \mathsf{weight}_{T,A,i_k})$.

Let us first consider an easy case for the fragment container unboundedness problem: If there are $A \in \mathsf{Act}$ with $\mathsf{Ctn}(A) = (i_1, \cdots, i_k)$ and a transition rule $A \xrightarrow{\mu} T$ such that $\mathsf{weight}_{T,A,i_j} > 0$ for some $j \in [k]$, then report "fragment container unbounded". The fragment container unboundedness in this case is attributed to the fact that the transition $A \xrightarrow{\mu} T$ can be applied for arbitrarily many times so that the height of fragment container i_j becomes unbounded.

Next, let us consider the general cases that there are no $A \in \mathsf{Act}$ and transition rule $A \xrightarrow{\mu} T$ satisfying the aforementioned condition, that is, for each $A \in \mathsf{Act}$ with $\mathsf{Ctn}(A) = (i_1, \cdots, i_k)$ and each transition rule $A \xrightarrow{\mu} T$, we have $\mathsf{weight}_{T,A,i_j} \leq 0$ for every $j \in [k]$.

For the general cases, we need to consider the transition rules of the form $F \xrightarrow{\mu} T$. Since the enablement of these transition rules depend on the top fragment of containers, it is necessary to determine the top fragments of containers. We use a tuple $(\mathsf{Frg} \cup \{\perp\})^k$ to denote the top fragments of containers, where $\perp$ denotes that the container is empty or the top fragment of the container cannot be determined. Suppose the current activity is A and its top fragments of containers are $(F_1, \cdots, F_k) \in (\mathsf{Frg} \cup \{\perp\})^k$. Then the top fragments of containers after applying T to A, denoted by $\mathsf{TopFrag}_T(F_1, \cdots, F_k)$, are defined as $(\mathsf{TopFrag}_{U_{T,A,i_1}}(F_1), \cdots, \mathsf{TopFrag}_{U_{T,A,i_k}}(F_k))$, where for every $j \in [k]$,

- if $U_{T,A,i_j} = \epsilon$, then $\mathsf{TopFrag}_{U_{T,A,i_j}}(F_j) = F_j$,
- otherwise, let the last action of U_{T,A,i_j} be $\beta'(F')$,
  - if $\beta' \neq \mathsf{REM}$ or $F' \neq F_j$, then $\mathsf{TopFrag}_{U_{T,A,i_j}}(F_j) = F'$,
  - otherwise, $\mathsf{TopFrag}_{U_{T,A,i_j}}(F_j) = \perp$. (If $\beta' = \mathsf{REM}$ and $F' = F_j$, we cannot determine the top fragment of container i_j.)

For each $A \in \mathsf{Act}$, we compute an edge-labled graph $G_A = (V, E)$ by executing the following procedure. Let $\mathsf{Ctn}(A) = (i_1, \cdots, i_k)$.

(1) Let G_0 be the graph comprising all the edges $v_0 \xrightarrow{\mathsf{weight}_{T,A}} (T, \mathsf{TopFrag}_T(\perp^k))$ such that $A \xrightarrow{\mu} T$ is a transition rule in $\mathcal{M}$ and $\mathsf{weight}_{T,A,i_j} \leq 0$ for every $j \in [k]$, where v_0 is a special vertex.
(2) Let $i = 0$. Iterate the following procedure until $G_{i+1} = G_i$: Obtain G_{i+1} from G_i by adding the following edges: For each vertex $(T, \vec{F})$ in G_i with $\vec{F} = (F_1, \cdots, F_k)$ and each transition rule $F_j \xrightarrow{\mu} T'$ such that $j \in [k]$ and $F_j \neq \perp$, add the edge $(T, \vec{F}) \xrightarrow{\mathsf{weight}_{T',A}} (T', \mathsf{TopFrag}_{T'}(\vec{F}))$.

Note that each edge of G_A is labeled by a tuple from $(\mathbb{N} \cup \{-\infty\})^k$.

A *witness cycle* of G_A for $A \in \mathsf{Act}$ is a cycle $(T_0, \overrightarrow{F_0}) \xrightarrow{\overrightarrow{w_1}} (T_1, \overrightarrow{F_1}) \cdots \xrightarrow{\overrightarrow{w_m}} (T_m, \overrightarrow{F_m})$ such that for some $j \in [k]$, $\sum_{r \in [m]} w_{r,j} > 0$, where $\overrightarrow{w_r} = (w_{r,1}, \cdots, w_{r,k})$ for each $r \in [m]$. Intuitively, $\sum_{r \in [m]} w_{r,j} > 0$ guarantees that the height of the container i_j is strictly increased after executing the all fragment transactions in the cycle. Note that in the definition of witness cycle, it is not required that $F_{r,j} \neq \perp$ for every $r \in [m]$ since the top fragments of the container i_j may not be used in the transitions and what is concerned is only the height increase of the container i_j.

Finally, the procedure is summarized as follows.

Algorithm to decide the fragment container unboundedness problem

If there are $A \in \mathrm{Act}$ with $\mathrm{Ctn}(A) = (i_1, \cdots, i_k)$ and a transition rule $A \xrightarrow{\mu} T$ such that $\mathrm{weight}_{T,A,i_j} > 0$ for some $j \in [k]$, then report "fragment container unbounded".
Otherwise, if there is $A \in \mathrm{Act}$ such that G_A contains a witness cycle, then report "fragment container unbounded".
Otherwise, report "unknown".

9 IMPLEMENTATION AND EVALUATION OF THE STATIC ANALYSIS ALGORITHMS

In this section, we implement the static analysis algorithms proposed in the previous section, giving rise to a tool TaskDroid[10]. Moreover, we evaluate the performance of TaskDroid via extensive experiments on 8,887 open-source and commercial Android apps.

9.1 Implementation

As depicted in Fig 12, TaskDroid comprises two modules, i.e., AMASSExtractor and AMASSAnalyzer.

- AMASSExtractor ($\mathrm{ICCBot}_{\mathrm{AMASS}}$) consists of two submodules, i.e., static AMASSExtractor and dynamic AMASSExtractor. Static AMSSExtractor statically builds AMASS models from Android apps. Moreover, as described in Section 5, for those apps whose AMASS models cannot be statically built, dynamic AMASSExtractor can build the AMASS models dynamically, by utilizing the UIAutomator and ADB tools. The inputs of AMASSExtractor are Android PacKage (APK) files.
- AMASSAnalyzer carries out static analysis on AMASS models. AMASSAnalyzer includes a submodule of *Unboundedness Analyzer*, which implements the procedure in Section 8. It also includes a submodule of *Reachability Analyzer*, which is used to generate a path starting from the main activity and a witnessing cycle following the path, when the AMASS is found to be task unbounded and fragment container unbounded.

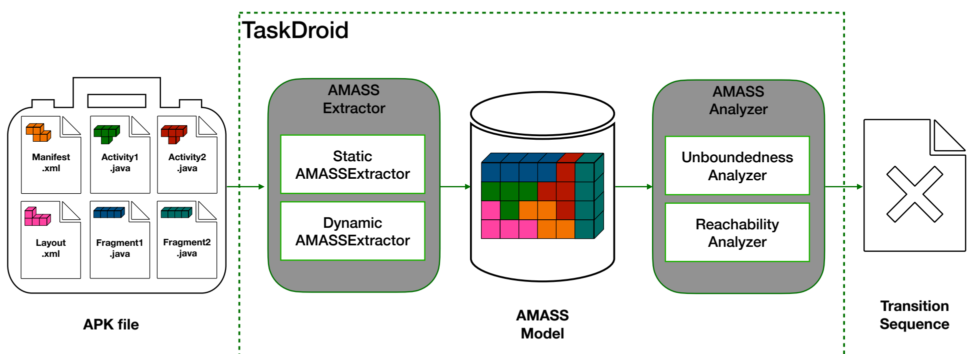

Fig. 12. Architecture of TaskDroid

[10]available at https://github.com/Jinlong-He/TaskDroid

9.2 Evaluation

We evaluate the efficiency and effectiveness of TaskDroid on 8,887 open source and commercial apps (cf. Section 9.2.1 and Section 9.2.2). Moreover, we evaluate the impact of different model extractors on the detection of task/fragment container unboundedness problems of these apps (Section 9.2.3).

The benchmark used for the evaluation consists of 8,887 Android apps collected from Androzoo[11] which include all the 3,867 open-source apps in the F-Droid repository,[12] as well as 5,020 commercial apps in the Google Play market. For Google Play, the apps are selected according to the popularity (number of downloads). Statistics of these apps can be found in Table 7.

Source	# apps	Avg/Max. size of apps	# extracted models (static/dynamic)
F-Droid	3,867	4.9M/285.0M	3,471 (2,580/891)
Google Play	5,020	15.7M/361.9M	4,612 (3,808/804)

Table 7. Statistics of benchmarks

AMASSExtractor extracts AMASS models from the APK files as described in Section 5.

- Out of the 3,867 F-Droid (resp. 5,020 Google Play) apps, static AMASSExtractor extracts 2,580 (resp. 3,808) AMASS models. Static AMASSExtractor fails on 1,287 F-Droid (resp. 1,212 Google Play) apps, which is attributed to the fact that Soot fails to decompile the APK files or the source codes of the apps are obfuscated or encrypted.
- Out of the 2,580 models that are extracted from the F-Droid apps (called F-Droid models for short) by static AMASSExtractor and 3,808 models that are extracted from the Google Play apps (called Google-Play models for short) by static AMASSExtractor, $\text{ICCBot}_{\text{AMASS}}$ discovers that 311 F-Droid models and 501 Google-Play models respectively involve cross-app ICCs. Note that although $\text{ICCBot}_{\text{AMASS}}$ discovers that these models involve cross-app ICCs, each of these models is extracted from *one* APK file and the cross-app ICCs therein have not been dealt with yet. To remedy this, we tried to manually identify the APK files involved in these cross-app ICCs by searching for their package names. In the end, for 125 F-Droid apps and 211 Google-Play apps respectively, we provided the missing APK files involved in the cross-app ICCs of these apps and successfully constructed the AMASS models for them where the cross-app ICCs are modeled. The average/maximum numbers and sizes of these missing apps involved in cross-app ICCs as well as the sizes of the extracted models are shown in Table 8. Afterwards, these models can be analyzed in the same way as a model that is extracted from one APK file.
- Moreover, among the 1,287 F-Droid (resp. 1,212 Google Play) apps that cannot be decompiled, dynamic AMASS extracts 891 (resp. 804) AMASS models. Dynamic AMASSExtractor fails on 396 F-Droid (resp. 408 Google Play) apps, which is attributed to the fact that some apps cannot be launched, or crash after launching, or require user name and password to proceed after launching.

9.2.1 The efficiency of TaskDroid. We investigate the efficiency and scalability of TaskDroid by examining all 3,471 F-Droid and 4,612 Google Play apps for which AMASS models are extracted. For each app, static AMASSExtractor extracts an AMASS model from the APK file statically, where the timeout is set to 300 seconds. Moreover, for the apps where static AMASSExtractor fails, dynamic AMASSExtractor extracts an AMASS model dynamically, where the time

[11] https://androzoo.uni.lu/
[12] https://f-droid.org/

Source	#apps containing cross-app ICCs	avg/max. numbers of missing apps	avg/max. sizes of missing apps	avg/max. sizes of models		
				|Act|	|Frg|	|Δ|
F-Droid	125	1.2/2	6.8M/103.2M	6.2/38	0.9/11	10.1/66
Google Play	804	1.5/2	16.1M/153.3M	7.9/19	1.1/20	14.5/97

Table 8. Statistics of the missing apps involved in cross-app ICCs

out is set to 600 seconds. The average/maximum sizes of the extracted models as well as the extraction time are shown in Table 9–10.

It can be observed that, for the statically extracted models, the average/maximum size (number of transition rules, activities, fragments) of F-Droid apps is much smaller than that of Google Play apps. This is expected, since the Google Play apps are considerably more complex than F-Droid open-source apps. Moreover, the average time spent in the static model extraction of Google Play apps is over 5 times more than that of F-Droid apps. Similar phenomenon happens for the dynamic model extraction, though the difference is slightly smaller.

Furthermore, static AMASSExtractor can extract the AMASS models for F-Droid apps in less than 1 minute and for Google Play apps in less than 4 mins, whereas dynamic AMASSExtractor can extract the AMASS models for F-Droid apps in around 5 mins and for Google Play apps in around 10 mins. The static model extraction is faster than the dynamic one, while the dynamic approach can deal with the apps that the static approach fails.

Source	Avg/Max. size of models			Avg. Time
	|Act|	|Frg|	|Δ|	
F-Droid	4.9/70	0.5/16	7.9/475	43.5s
Google Play	7.1/509	1.0/121	21.0/1,956	234.6s

Table 9. Static AMASSExtractor

Source	Avg/Max. size of AMASS		Avg. Time
	|Act|	|Δ|	
F-Droid	6.1/31	15.3/71	313.4s
Google Play	11.2/43	28.1/129	593.2s

Table 10. Dynamic AMASSExtractor

To detect task/fragment container unboundedness vulnerabilities, the timeout is set to 30 seconds, the height $\hbar$ of tasks is set to 6 and k (i.e., the number of interplaying tasks, cf. Section 8) is set to 2.

We first consider Android 13.0. The results of task unboundedness (resp. fragment container unboundedness) are presented in Table 11–12. As one may see, TaskDroid is highly efficient in analyzing the AMASS models, which can be done in less than 0.1 seconds per app on average. Moreover, TaskDroid identifies 485 (14.0%) F-Droid and 559 (12.1%) Google Play models as "task unbounded". On the other hand, TaskDroid identifies 193 (5.6%) F-Droid and 293 (6.4%) Google Play models as "fragment container unbounded". Finally, the number of witnessing cycles discovered for the unbounded models is around 3 per app on average.

We then consider Android 6.0–12.0 and compare the results for these versions with those for Android 13.0. As shown in Figure 13, the numbers of task unbounded F-Droid (resp. Google Play) apps are slightly different for different versions,

Source	#Unbounded models	#witnessing cycles per model	Time(s)
F-Droid	485/3,471 (14.0%)	3.3	0.05s
Google Play	559/4,612 (12.1%)	3.9	0.1s

Table 11. Task Unboundedness

Source	#Unbounded models	#witnessing cycles per model	Time(s)
F-Droid	193/2,580 (7.5%)	2.5	0.05
Google Play	293/3,808 (7.7%)	3.0	0.1

Table 12. Fragment Container Unboundedness

while the numbers of fragment container unbounded F-Droid (resp. Google Play) apps are the same for different versions. This phenomenon is explained by that the semantics of AMASS$_{ACT}$ is slightly different across Android versions, while the semantics of AMASS$_{FRG}$ is the same for all these versions. To have a closer look, we can see that the numbers of task unbounded F-Droid (resp. Google Play) apps for Android 7.0, Android 8.0–10.0, and Android 11.0–13.0 are close to each other, and are slightly away from that of Android 6.0. This is explained by that Android 6.0 uses a different task allocation mechanism from the others.

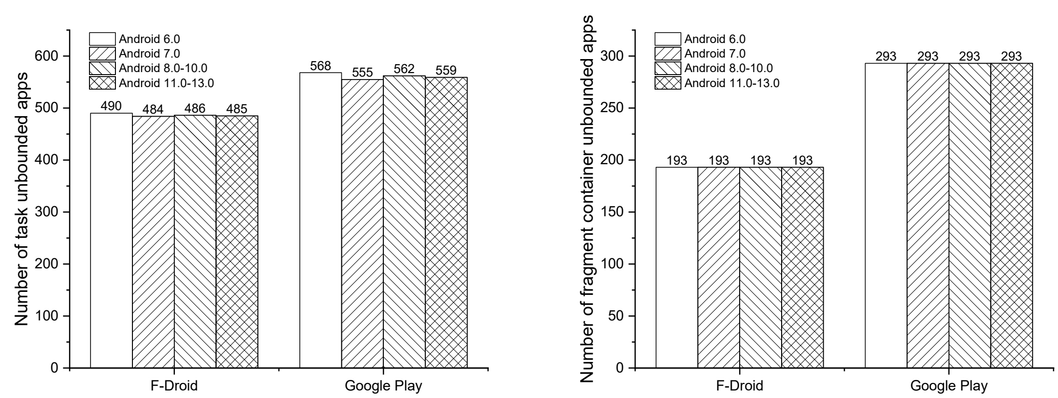

Fig. 13. Comparison of the results for different Android versions

9.2.2 The effectiveness of TaskDroid. We investigate whether TaskDroid can accurately detect task/fragment container unboundedness vulnerabilities, and the abnormal behavior caused by these vulnerabilities in Android mobile device. To this end, we randomly select 50 apps from those that are identified as "task unbounded" and "fragment container unbounded" by TaskDroid respectively, in each case 25 are from F-Droid and 25 are from Google Play. The information of these apps can be found in Appendix G, Table 20 – 23.

We manually check on a virtual machine (4-core CPU, 2G RAM with Android 13.0) whether these 100 apps are indeed task/fragment-container unbounded by verifying whether the witnessing cycles reported by TaskDroid are executable. To ensure the reliability of the manual checking, we ask 4 individuals to complete the manual checking and each of them checks all the 100 apps. The results are cross checked, and summarized in Table 13.

Source	unbound. vulnerability	#apps	#confirmed apps
F-Droid	task	25	17 (68%)
Google Play	task	25	16 (64%)
F-Droid	fragment container	25	16 (64%)
Google Play	fragment container	25	16 (64%)
F-Droid or Google Play	task	50	33 (66%)
F-Droid or Google Play	fragment container	50	32 (64%)
F-Droid or Google Play	task or fragment container	100	65 (65%)

Table 13. Manual confirmation of the effectiveness of TaskDroid

Out of the 25 apps from F-Droid (resp. Google Play), 17 (resp. 16) apps are confirmed to be task unbounded, occupying 68% (resp. 64%) of the randomly chosen apps; 16 (resp. 16) apps are confirmed to be fragment-container unbounded, occupying 64% (resp. 64%) of the randomly chosen F-Droid (resp. Google Play) apps. These give an overall accuracy of 65%. For the rest 35 apps, we could not confirm the analysis result, because: 9 apps require login, 9 apps crash immediately after launching, and 17 apps could not execute the witnessing cycles (we suspect that the witnessing cycles in these AMASS models are spurious).

Furthermore, on the same virtual machine, for each app that is confirmed to be "task unbounded" or "fragment-container unbounded", we use UIAutomator to execute the click-event sequence generated when simulating the witnessing cycle for many times until some abnormal behavior appears. During the execution, we use ADB to calculate the number of repetitions of the witnessing cycle as well as record the type of abnormal behavior of the virtual device. The results of the experiments are presented in Table 14–17.

- Among the 18 (resp. 16) F-Droid (resp. Google Play) apps that are confirmed to be task unbounded, after hundreds of repetitions of the witnessing cycle, 10 (resp. 11) apps end up with rebooting of device, 6 (resp. 4) apps end up with app crash, and 2 (resp. 1) apps end up with restarting of app.
- Among the 15 (resp. 14) F-Droid (resp. Google Play) apps that are confirmed to be fragment-container unbounded, after thousands of repetitions of the witnessing cycle, 2 (resp. 5) apps end up with rebooting of device, 10 (resp. 8) apps end up with app crash, and 3 (resp. 1) apps end up with restarting of app.

To further evaluate the threat of task unboundedness for the popular commercial apps, we select 9 apps from Google Play with more than 100,000 downloads, including well-known Amazon, Netflix, and Youtube, Facebook apps. We build AMASS models of these apps dynamically, and carry out static analysis. We find (with confirmation) that 9 apps suffer

Package name	#repetitions of the witnessing cycle	Abnormal behavior
org.gnucash.android	215	reboot
systems.byteswap.aiproute	203	reboot
max.music_cyclon	97	reboot
com.matburt.mobileorg	204	reboot
org.npr.android.news	192	reboot
com.commonsware.android.arXiv	125	reboot
net.mabako.steamgifts	103	reboot
com.app.Zensuren	114	reboot
uk.co.busydoingnothing.prevo	122	reboot
de.drhoffmannsoftware.calcvac	208	reboot
org.sasehash.burgerwp	240	app crash
com.mikifus.padland	59	app crash
org.evilsoft.pathfinder.reference	179	app crash
com.android.keepass	150	app crash
de.bloosberg.basti.childresuscalc	288	app crash
com.samebits.beacon.locator	178	app crash
ohm.quickdice	169	app restart

Table 14. Threats of task unboundedness vulnerabilities to F-Droid apps

Package name	#repetitions of the witnessing cycle	Abnormal behavior
com.abdulqawi.ali.mosabqa	288	reboot
com.music.star.player	189	reboot
com.drclabs.android.wootchecker	351	reboot
com.hotels.hotelsmecca	169	reboot
com.airg.hookt	137	reboot
com.holidu.holidu	197	reboot
com.appkey.english3000freekata	113	reboot
socials.com.application	99	reboot
www.genting.rwgenting	114	reboot
com.goldenhammer.beisboldominicana	125	reboot
com.travolution.seoultravelpass	144	reboot
com.ic.myMoneyTracker	179	app crash
tekcarem.gebeliktakibi	161	app restart
de.fckoeln.app	111	app restart
de.twokit.castbrowser	223	app restart
com.TWTD.FLIXMOVIE	139	app restart

Table 15. Threats of task unboundedness vulnerabilities to Google Play apps

from task unboundedness vulnerability, which, after less than one hundred of repetitions the witnessing cycles, all end up with rebooting of device (cf. Table 18 for details).

Considering that the crashes and reboots of mobile applications can be caused by a variety of reasons and it is common for mobile apps to experience such issues, to demonstrate the causality between the crashes and task/fragment-container

Package name	#repetitions of the witnessing cycle	Abnormal behavior
org.ligi.fahrplan	1187	reboot
io.gitlab.allenb1.todolist	1269	reboot
naman14.timber	815	app crash
me.anon.grow	743	app crash
com.wikijourney.wikijourney	932	app crash
com.mattallen.loaned	641	app crash
com.ymber.eleven	1032	app crash
com.syncedsynapse.kore2	952	app crash
net.momodalo.app.vimtouch	993	app crash
com.csipsimple	1145	app crash
ch.corten.aha.worldclock	899	app crash
org.wikimedia.commons.wikimedia	1002	app crash
com.llamacorp.equate	1145	app restart
koeln.mop.elpeefpe	1137	app restart
fr.kwiatkowski.ApkTrack	1257	app restart
eu.prismsw.lampshade	139	app restart

Table 16. Threats of fragment container unboundedness vulnerabilities to F-Droid apps

Package name	#repetitions of the witnessing cycle	**Abnormal behavior**
com.schoola2zlive	887	reboot
com.endless.smoothierecipes	734	reboot
com.traderumors	995	reboot
com.rakuten.room	873	reboot
com.hotels.hotelsmecca	976	reboot
br.com.prevapp03	671	app crash
com.star.mobile.video	1056	app crash
fr.elol.yams	990	app crash
music.symphony.com.materialmusicv2	1234	app crash
ru.sports.rfpl	891	app crash
com.discsoft.daemonsync	795	app crash
com.accuvally.android.accupass	931	app crash
com.directv.navigator	722	app crash
kvp.jjy.MispAndroid320	899	app crash
de.wirfahrlehrer.easytheory	971	app crash
com.ldf.gulli.view	1103	app restart

Table 17. Threats of fragment container unboundedness vulnerabilities to Google Play apps

unboundedness, we use the Monkey tool[13] to stress-test the 74 apps in Table 14—18. We set the timeout period of Monkey to be 30 minutes. As shown in Figure 14, Monkey reports 6 apps ending up with app crash, 4 apps ending up with app restart, and 4 apps ending up with rebooting of device. Namely, 14 out of 74 apps (around 19%) apps end up

[13] https://developer.android.com/studio/test/other-testing-tools/monkey

Package name (size)	**#repetitions of the witnessing cycle**	Abnormal behavior
com.taobao.taobao (151.0)	66	reboot
com.jingdong.app.mall (92.6)	59	reboot
com.amazon.mShop.android.shopping (73.8)	73	reboot
com.contextlogic.wish (21.9)	63	reboot
com.google.android.youtube (121.3)	54	reboot
com.netflix.ninja (100.6)	76	reboot
com.instagram.android (49.2)	89	reboot
com.zhiliaoapp.musically (167.8)	65	reboot
com.facebook.katana (74.2)	96	reboot

Table 18. Threats of task unboundedness vulnerabilities to popular commercial apps

with abnormal behaviors which shows that, compared to the random testing, the repeated executions of witnessing cycles indeed dramatically increase the chances of exposing abnormal behaviors.

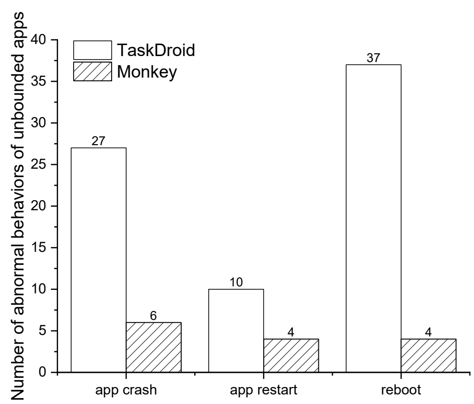

Fig. 14. Number of abnormal behaviors of unbounded apps by TaskDroid and Monkey

Finally, we demonstrate the causality between the crashes and task/fragment-container unboundedness by analyzing the root causes of the reported crashes.

At first, we use the "logcat" command in ADB to dump the logs of system messages when the witnessing cycles of the apps in Table 14—18 are executed repeatedly. From the logs, we discover that

- when the abnormal behaviors appear, all the apps with the app-crash behavior report the error message "SurfaceFlinger: AddClientLayer failed, mNumLayers (4096) >= MAX_LAYERS (4096)", and
- all the apps with the app-restart or reboot behavior report the exception "android.os.DeadObjectException".

We inspect the Android OS documentation as well as the source code, in order to understand the meanings of the errors/exceptions.

- From the Android OS source code, we know that the error message “SurfaceFlinger: AddClientLayer failed, mNumLayers (4096) >= MAX_LAYERS (4096)” is produced when the number of layers in an app goes beyond the limit (i.e. 4096), where a layer is a combination of a surface and an instance of the class SurfaceControl.
- On the other hand, from the Android OS documentation, we know that the “android.os.DeadObjectException” means that “The object you are calling has died, because its hosting process no longer exists, or there has been a low-level binder error”.

Then we use the “adb shell dumpsys meminfo” command to monitor the memory usage of Android apps when the witnessing cycles are executed repeatedly and discover that the memory sizes grow monotonically. This indicates that the abnormal behaviors are resulted from the memory issues. In particular, when the memory is used up, either the Android OS kills the process for the app and restarts the app, or even the proper running of the system processes (e.g. the system launcher and system services) is affected and the Android OS may reboot. Finally, to understand why different abnormal behaviors appear when the witnessing cycles of different apps are executed repeatedly, we extract the information about the growth of the memory usage when the witnessing cycle is executed *once*, which can be found in Table 19. From Table 19, we can see that the average memory usage per witnessing-cycle execution for app-crash is much smaller than that for app-restart or reboot. As a result, for app-crash, because the execution of the witnessing cycle occupies a small amount of memory, the memory is *not* used up before reaching the limit on the number of layers. Therefore, in the end, the error “SurfaceFlinger: AddClientLayer failed” is reported and the app crashes. This (partially) explains why different abnormal behaviors appear.

unbound. vulnerability	**abnormal behavior**	**avg. memory usage per witnessing-cycle execution**
task	app crash	1.2M
	app restart	10.6M
	reboot	12.7M
fragment container	app crash	0.6M
	app restart	3.4M
	reboot	3.9M

Table 19. Average memory usage per witnessing-cycle execution

9.2.3 Comparison of the impact of different model extractors. We compare the impact of the following three model extractors on the effectiveness of the task/fragment container unboundedness problems: $\text{ICCBot}_{\text{AMASS}}$, ICCBot, and ActExtractor, where ActExtractor is the model extractor from [HCW+19], where only activities were considered while fragments were ignored.

We first run experiments to compare the capabilities of the three extractors to construct AMASS models. The experiment results are in Figure 15, which include the numbers of models constructed by these extractors and the average number of transition rules in these models. From Figure 15, we can see that $\text{ICCBot}_{\text{AMASS}}$ constructs more models than the other two extractors, since it is the only extractor that is able to construct models dynamically from APK files. Moreover, the average number of transition rules in the models constructed by $\text{ICCBot}_{\text{AMASS}}$ is greater than the other two extractors, which is mainly because the models constructed dynamically by $\text{ICCBot}_{\text{AMASS}}$ include much more transition rules than those constructed statically, since these apps are the ones that cannot be decompiled and normally large commercial apps.

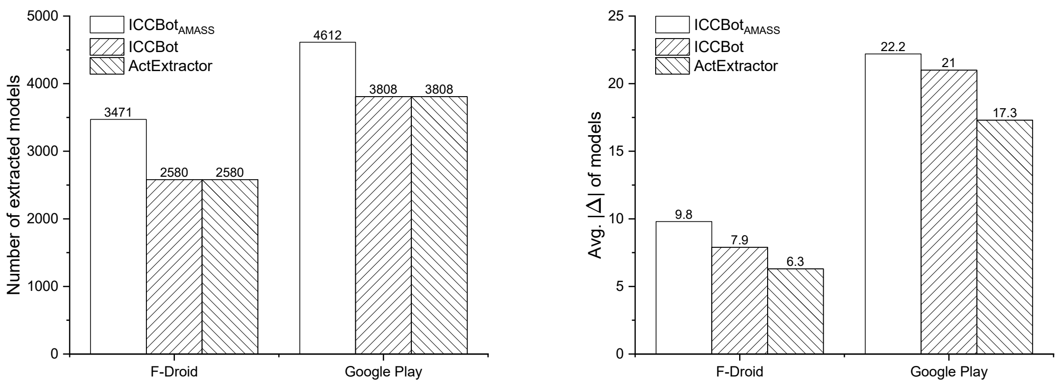

Fig. 15. Numbers of models extracted by different extractors and the average number of transitions in these models

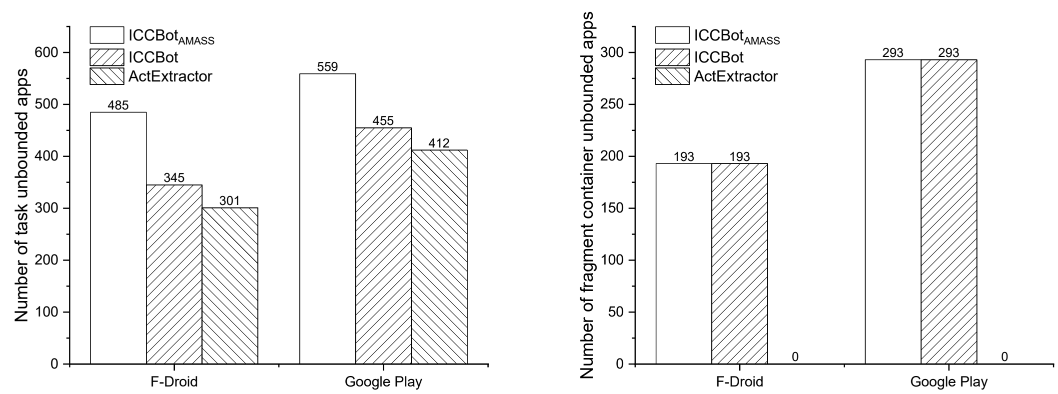

Fig. 16. Numbers of task/fragment container unbounded apps using different extractors

To compare the impact of different extractors on TaskDroid, for each app, we run AMASSAnalyzer on the three models constructed by the three extractors for the app. We summarize the results in Figure 16, where we can see that the number of task unbounded apps is the greatest when ICCBot$_{\text{AMASS}}$ is used. This is because ICCBot$_{\text{AMASS}}$ extracts more models than the other two extractors since it utilizes the dynamic approach to extract the models from the commercial apps that cannot be decompiled. In Table 18, by repeatedly executing the witnessing cycles, we confirm that among these apps whose models are constructed dynamically by ICCBot$_{\text{AMASS}}$, 9 apps are indeed task unbounded. Note that ICCBot and ActExtractor are unable to construct models from these apps since their APK files cannot be decompiled successfully by Soot. On the other hand, TaskDroid reports the same number of fragment container unbounded apps for ICCBot$_{\text{AMASS}}$ and ICCBot. This is due to the fact that although ICCBot$_{\text{AMASS}}$ extracts more information about fragments than ICCBot, e.g., the API addToBackstack(TS), the information has no impact on our static analysis algorithm for the fragment container unboundedness problem (see Section 8). Moreover, no fragment unboundedness is reported for the models extracted by ActExtractor, since it only extracts the information for activities.

9.3 Validation of the "small number of tasks" and "small height of stacks" hypotheses

We validate the "small number of tasks" and "small height of stacks" hypotheses used in Section 8. Recall that for the analysis of task unboundedness in Section 8, we hypothesize that only a small number k of tasks are involved.

We first evaluate the validity of the "small number of tasks" hypothesis for task unboundedness by varying k from 0 to 3 and checking the growth of the number of task unbounded apps detected by TaskDroid. The experimental results are shown in Figure 17. We observe that when k is increased from 1 to 2, the number of task-unbounded apps increases only slightly. Moreover, when k is increased from 2 to 3, the number of task-unbounded apps keeps unchanged. This suggests that, in order to identify those task-unbounded apps, a small k suffices (in this case $k = 2$). This also justifies our choice of k in the static analysis of task unboundedness for AMASS models. This phenomenon is explained by the observation that for a majority of apps, the number of task affinities of their activities is small (or even equal to 1).

Moreover, in Section 8, the height bound $\hbar = 6$ of stacks was used to generate the witnessing transition sequences for task unbounded and fragment container unbounded AMASS models. We then carry out experiments to validate this "small height of stacks" hypothesis. The experiments proceed as follows: we first set $\hbar = 1$, use nuXmv to generate the witnessing transition sequences for the AMASS models that are identified as "task unbounded" by TaskDroid, and count the number of models for which the witnessing transition sequences can be successfully generated. Then we experiment $\hbar = 2$–7. The results are summarized in Figure 18, where we can see that the number of models for which the witnessing transition sequences can be successfully generated grow when $\hbar$ is increased from 1 to 6, while keep unchanged when $\hbar$ is increased from 6 to 7. The results justify the "small height of stacks" hypothesis and our choice of $\hbar = 6$ in Section 8.

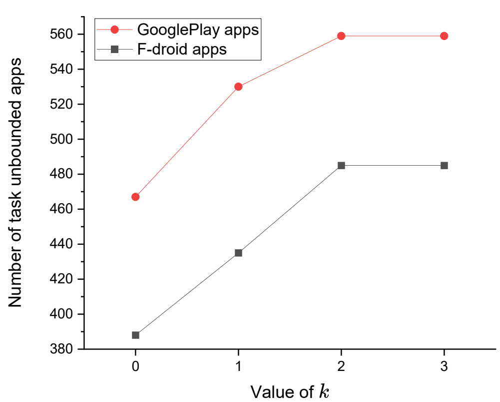

Fig. 17. "Small number of tasks" hypothesis: k

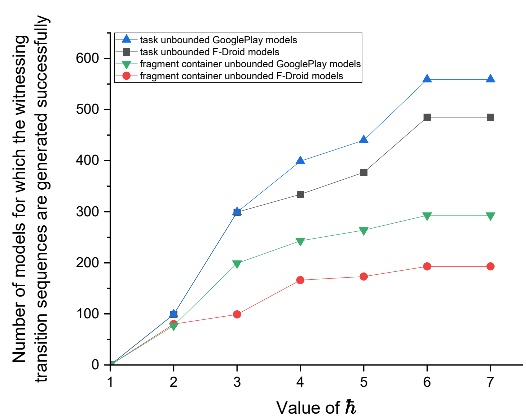

Fig. 18. "Small height of stacks" hypothesis: $\hbar$

10 CONCLUSION

In this paper, we have formalized the semantics of the Android multitasking mechanism by a new model AMASS, which have been validated against the actual behaviors of Android systems. Based on the semantics, we provide new static analysis algorithms for detecting potential task unboundedness and fragment container unboundedness vulnerabilities. We have implemented a static analysis tool TaskDroid. The experiments show that TaskDroid is able to discover the task-unboundedness and fragment-container-unboundedness vulnerabilities for many open-source and commercial Android apps, which can be exploited to produce abnormal behaviors, e.g. black screen, app crash or even device reboot.

The formal semantics of the AMASS model defined in this paper is valuable for both the developers of Android apps and the researchers on the analysis and testing of Android apps.

- the developers can read the concise and precise semantics of the AMASS model, instead of the source code of the Android OS, to understand the Android activity-fragment multitasking mechanism.
- Compared to the various models (e.g. activity transition graphs) in literature, the AMASS model provides refined modeling of the Android activity-fragment multitasking mechanism, and can be utilized to improve the accuracy of the analysis and testing of Android apps.
- The task unboundedness issue identified in this paper is a novel type of security threats for Android apps, contributing to the understanding the security aspect of Android UI design.

For the future work, more problems in static analysis can benefit from the formalized multi-tasking semantics. Moreover, we believe that AMASS is a fundamental model in Android UI research which deserves a thorough theoretical investigation.

REFERENCES

[AN13] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 641–660. ACM, 2013.

[CCD+14a] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 334–342, 2014.

[CCD+14b] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[CHGD18] Jia Chen, Ge Han, Shanqing Guo, and Wenrui Diao. Fragdroid: Automated user interface interaction with activity and fragment analysis in android applications. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 398–409. IEEE Computer Society, 2018.

[CHS+18] Taolue Chen, Jinlong He, Fu Song, Guozhen Wang, Zhilin Wu, and Jun Yan. Android stack machine. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 487–504. Springer, 2018.

[HCW+19] Jinlong He, Taolue Chen, Ping Wang, Zhilin Wu, and Jun Yan. Android multitasking mechanism: Formal semantics and static analysis of apps. In Anthony Widjaja Lin, editor, *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, volume 11893 of *Lecture Notes in Computer Science*, pages 291–312. Springer, 2019.

[int23] https://developer.android.com/reference/android/content/Intent#flags, 2023.

[LBP+17] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.*, 88:67–95, 2017.

[LHR17] Sungho Lee, Sungjae Hwang, and Sukyoung Ryu. All about activity injection: threats, semantics, and detection. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 252–262. IEEE Computer Society, 2017.

[LLW+12] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: statically vetting android apps for component hijacking vulnerabilities. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 229–240. ACM, 2012.

[LWX18] Jie Liu, Diyu Wu, and Jingling Xue. Tdroid: exposing app switching attacks in android with control flow specialization. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 236–247. ACM, 2018.

[OLD+15] Damien Octeau, Daniel Luchaup, Matthew L. Dering, Somesh Jha, and Patrick D. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 77–88. IEEE Computer Society, 2015.

[OMJ+13] Damien Octeau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 543–558. USENIX Association, 2013.

[YWYZ17] Jiwei Yan, Tianyong Wu, Jun Yan, and Jian Zhang. Widget-sensitive and back-stack-aware GUI exploration for testing android apps. In *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 42–53, 2017.

[YZL+22a] Jiwei Yan, Shixin Zhang, Yepang Liu, Xi Deng, Jun Yan, and Jian Zhang. A comprehensive evaluation of android icc resolution techniques. In *ASE 2022*, 2022.

[YZL+22b] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. Iccbot: Fragment-aware and context-sensitive ICC resolution for android applications. In *44th 2022 IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 105–109. IEEE, 2022.

[YZW+15] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. Static window transition graphs for android. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 658–668, 2015.

[ZSX18] Yifei Zhang, Yulei Sui, and Jingling Xue. Launch-mode-aware context-sensitive activity transition analysis. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 598–608. ACM, 2018.

A SEMANTICS OF $A \xrightarrow{\mathsf{finishStart}(\phi)} B$ FOR AMASS$_{ACT,LM}$

In the sequel, assuming that $\mathcal{M}$ is an AMASS$_{ACT,LM}$.

Let $\rho = (\Omega_1, \cdots, \Omega_n)$ be a configuration with $\Omega_i = (S_i, A_i, \zeta_i)$ for each $i \in [n]$, and $S = [B_1, \cdots, B_m]$ be a task. The following additional auxiliary function $\mathsf{RmAct}(\rho, i, j)$ is defined for finishStart.

- let $1 \le i \le n$, $S_i = [C_1, \cdots, C_l]$ and $1 \le j \le l$, then $\mathsf{RmAct}(\rho, i, j) = (\Omega_1, \cdots, \Omega_{i-1}, (S_i', A_i, \zeta_i), \Omega_{i+1}, \cdots, \Omega_n)$ if $l > 1$, where $S_i' = [C_1, \cdots, C_{j-1}, C_{j+1}, \cdots, C_l]$, and $\mathsf{RmAct}(\rho, i, j) = (\Omega_1, \cdots, \Omega_{i-1}, \Omega_{i+1}, \cdots, \Omega_n)$ otherwise.

We present the semantics of the transition rules $A \xrightarrow{\mathsf{finishStart}(\phi)} B$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{STD}}$

- If $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, then $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$.
- If $\mathsf{Lmd}(A) = \mathsf{SIT}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq \mathsf{MAIN}$, then $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i = \mathsf{MAIN}$, or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$, then $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
  - if $\mathsf{GetTsk}(\rho, B) = *$, then $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{STK}), 2, 1)$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{STP}}$

- If $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, then
  - if $A = B$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$,
  - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$.
- If $\mathsf{Lmd}(A) = \mathsf{SIT}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq \mathsf{MAIN}$, then $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i = \mathsf{MAIN}$, or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$,
    - if $\mathsf{TopAct}(S_i) = B$, then $\rho' = \mathsf{RmAct}(\mathsf{MvAct2Top}(\rho, i), 2, 1)$,
    - otherwise $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
  - if $\mathsf{GetTsk}(\rho, B) = *$, then $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{STK}), 2, 1)$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{SIT}}$

- If $\mathsf{GetRealTsk}(\rho, B) = S_1$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$.
- If $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $i > 1$, then $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$.
- If $\mathsf{GetRealTsk}(\rho, B) = *$, then $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{SIT}), 2, 1)$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{STK}}$

- If $\mathsf{GetRealTsk}(\rho, B) = S_i$, or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$ then
  - if $i = 1$,
    - if $B \notin S_i$, then $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
    - if $B \in S_i$,
      - if $\mathsf{TopAct}(\rho) = B$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$,
      - if $\mathsf{TopAct}(\rho) \neq B$, then $\rho' = \mathsf{ClrTop}(\rho, B)$,
  - if $i > 1$,
    - if $B \notin S_i$, then $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
    - if $B \in S_i$, then $\rho' = \mathsf{RmAct}(\mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$.

- If $\mathrm{GetTsk}(\rho, B) = *$, then $\rho' = \mathrm{RmAct}(\mathrm{NewTsk}(\rho, B, \mathrm{STK}), 2, 1)$.

B SEMANTICS OF AMASS$_{ACT,IF}$

In the sequel, assuming that $\mathcal{M}$ is an AMASS$_{ACT,IF}$. Firstly, we need adapt the concept of configurations as follows: A configuration of $\mathcal{M}$ is encoded as a pair (ρ, b), where $b \in \{\mathrm{NOH}, \neg\mathrm{NOH}\}$ and $\rho = (\Omega_1, \cdots, \Omega_n)$ such that for each $i \in [n]$, $\Omega_i = (S_i, A_i, \zeta_i)$, where $S_i \in \mathrm{Act}^*$ is a task, $A_i \in \mathrm{Act}$ is the real activity of S_i, but $\zeta_i \in \{\mathrm{MAIN}, \mathrm{NTK}, \mathrm{NDM}\}$. Intuitively, NTK in ζ_i plays the same role as STK in AMASS$_{ACT,LM}$, and NDM is added for the intent flag NDM, moreover, SIT disappears since in AMASS$_{ACT,IF}$, the launch modes of all activities are assumed to be STD.

Let us define the semantics of $\mathcal{M}$ by a relation $\rho \xrightarrow[\tau]{\mathcal{M}} \rho'$ with $\tau = A \xrightarrow{\alpha(\phi)} B$. Let us consider the subcase $\phi \models \neg\mathrm{TOH}$ first.

$\boxed{\phi \models \neg\mathrm{NTK} \wedge \neg\mathrm{NDM}}$

- If $\phi \models \mathrm{CTP}$ and $B \in \mathrm{TopTsk}(\rho)$, then
  - if $\mathrm{TopAct}(\rho) \neq B$, then $\rho' = \mathrm{ClrTop}(\rho, B)$, moreover,
    - if $\phi \models \neg\mathrm{STP}$, then $b' = \mathrm{NOH}$ iff $\phi \models \mathrm{NOH}$,
    - otherwise, $b' = \neg\mathrm{NOH}$,
  - if $\mathrm{TopAct}(\rho) = B$,
    - if $\phi \models \mathrm{STP}$, then
      - if $\alpha = \mathrm{start}$, then $\rho' = \rho$ and $b' = b$,
      - if $\alpha = \mathrm{finishStart}$, then $\rho' = \mathrm{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathrm{NOH}$,
    - if $\phi \models \neg\mathrm{STP}$, then $\rho' = \rho$ and $b' = \mathrm{NOH}$ iff $\phi \models \mathrm{NOH}$.
- If $\phi \models \mathrm{CTP}$ and $B \notin \mathrm{TopTsk}(\rho)$, then $b' = \mathrm{NOH}$ iff $\phi \models \mathrm{NOH}$, moreover,
  - if $b = \neg\mathrm{NOH}$ and $\alpha = \mathrm{start}$, then $\rho' = \mathrm{Push}(\rho, B)$,
  - otherwise, $\rho' = \mathrm{RmAct}(\mathrm{Push}(\rho, B), 1, 2)$.
- If $\phi \models \neg\mathrm{CTP}$, then
  - if $\phi \models \mathrm{RTF}$ and $B \in \mathrm{TopTsk}(\rho)$, then
    - if $\mathrm{TopAct}(\rho) \neq B$, then $b' = \neg\mathrm{NOH}$, moreover,
      - if $b = \neg\mathrm{NOH}$ and $\alpha = \mathrm{start}$, then $\rho' = \mathrm{MvAct2Top}(\rho, B)$,
      - otherwise, $\rho' = \mathrm{RmAct}(\mathrm{MvAct2Top}(\rho, B), 1, 2)$,
    - if $\mathrm{TopAct}(\rho) = B$,
      - if $\alpha = \mathrm{start}$, then $\rho' = \rho$ and $b' = b$,
      - if $\alpha = \mathrm{finishStart}$, then $\rho = \mathrm{RmAct}(\rho, 1, 1)$, and $b' = \neg\mathrm{NOH}$,
  - if $\phi \models \mathrm{RTF}$ and $B \notin \mathrm{TopTsk}(\rho)$, then $b' = \mathrm{NOH}$ iff $\phi \models \mathrm{NOH}$, moreover,
    - if $b = \neg\mathrm{NOH}$ and $\alpha = \mathrm{start}$, then $\rho' = \mathrm{Push}(\rho, B)$,
    - otherwise, $\rho' = \mathrm{RmAct}(\mathrm{Push}(\rho, B), 1, 2)$.
  - If $\phi \models \neg\mathrm{RTF}$, then
    - if $\phi \models \mathrm{STP}$ and $\mathrm{TopAct}(\rho) = B$ or $\phi \models \mathrm{STP} \wedge \mathrm{PIT}$ and $\mathrm{PreAct}(\rho) = B$,
      - if $\alpha = \mathrm{start}$, then $\rho' = \rho$ and $b' = b$,
      - if $\alpha = \mathrm{finishStart}$, then $\rho' = \mathrm{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathrm{NOH}$,
    - otherwise, $b' = \mathrm{NOH}$ iff $\phi \models \mathrm{NOH}$, moreover,
      - if $b = \neg\mathrm{NOH}$ and $\alpha = \mathrm{start}$, then $\rho' = \mathrm{Push}(\rho, B)$,

· otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$.

$\boxed{\phi \models \mathsf{NDM}}$

- If $\phi \models \mathsf{MTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
  - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NDM})$,
  - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NDM}), 2, 1)$.
- If $\phi \models \neg\mathsf{MTK}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$, then
    - ∗ if $i \neq 1$, then
      - · if $\phi \models \neg\mathsf{CTK}$, then
        - ⋄ if $B \notin S_i$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
          - ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
          - ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
        - ⋄ otherwise, $b' = \neg\mathsf{NOH}$, moreover,
          - ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B)$,
          - ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
      - · if $\phi \models \mathsf{CTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
        - ⋄ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B)$,
        - ⋄ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
    - ∗ otherwise ($i = 1$),
      - · if $\phi \models \neg\mathsf{CTK}$, then
        - ⋄ if $B \notin S_1$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
          - ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
          - ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
        - ⋄ if $B \in S_1$ and $\mathsf{TopAct}(S_1) \neq B$, then $\rho' = \mathsf{ClrTop}(\rho, B)$ and $b' = \neg\mathsf{NOH}$,
        - ⋄ if $B \in S_1$ and $\mathsf{TopAct}(S_1) = B$ (this implies $A = B$), then
          - ∘ if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
          - ∘ if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
      - · if $\phi \models \mathsf{CTK}$, then $\rho' = \mathsf{ClrTsk}(\rho, B)$, and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
  - if $\mathsf{GetRealTsk}(\rho, B) = *$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
    - ∗ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NDM})$,
    - ∗ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NDM}), 2, 1)$.

$\boxed{\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM}}$

- If $\phi \models \mathsf{MTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
  - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NTK})$,
  - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NTK}), 2, 1)$.
- If $\phi \models \neg\mathsf{MTK}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$, then
    - ∗ if $i \neq 1$, then
      - · if $\phi \models \mathsf{CTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,

- ◇ if $b = \neg$NOH and $\alpha =$ start, then $\rho' = \text{ClrTsk}(\text{MvTsk2Top}(\rho, i), B)$,
- ◇ otherwise, $\rho' = \text{RmAct}(\text{ClrTsk}(\text{MvTsk2Top}(\rho, i), B), 2, 1)$,

· if $\phi \models \neg$CTK, then
- ◇ if $\phi \models$ CTP and $B \in S_i$, then
  - ○ if $b = \neg$NOH and $\alpha =$ start, then $\rho' = \text{ClrTop}(\text{MvTsk2Top}(\rho, i), B)$,
  - ○ otherwise, $\rho' = \text{RmAct}(\text{ClrTop}(\text{MvTsk2Top}(\rho, i), B), 2, 1)$,

  moreover,
  - ○ if $\phi \models \neg$STP, then $b' =$ NOH iff $\phi \models$ NOH,
  - ○ otherwise $b' = \neg$NOH,
- ◇ if $\phi \models$ CTP and $B \notin S_i$, then $b' =$ NOH iff $\phi \models$ NOH, moreover,
  - ○ if $b = \neg$NOH and $\alpha =$ start, then $\rho' = \text{Push}(\text{MvTsk2Top}(\rho, i), B)$,
  - ○ otherwise, $\rho' = \text{RmAct}(\text{Push}(\text{MvTsk2Top}(\rho, i), B), 2, 1)$,
- ◇ if $\phi \models \neg$CTP, then
  - ○ if $\phi \models$ RTF and $B \in S_i$, then $b' = \neg$NOH, moreover,
    - ★ if $b = \neg$NOH and $\alpha =$ start, then $\rho' = \text{MvAct2Top}(\text{MvTsk2Top}(\rho, i), B)$,
    - ★ otherwise, $\rho' = \text{RmAct}(\text{MvAct2Top}(\text{MvTsk2Top}(\rho, i), B), 2, 1)$,
  - ○ if $\phi \models$ RTF and $B \notin S_i$, then $b' =$ NOH iff $\phi \models$ NOH, moreover,
    - ★ if $b = \neg$NOH and $\alpha =$ start, then $\rho' = \text{Push}(\text{MvTsk2Top}(\rho, i), B)$,
    - ★ otherwise, $\rho' = \text{RmAct}(\text{Push}(\text{MvTsk2Top}(\rho, i), B), 2, 1)$,
  - ○ if $\phi \models \neg$RTF, then
    - ★ if $\text{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq$ MAIN, then $b' = \neg$NOH, moreover,
      - ▹ if $b = \neg$NOH and $\alpha =$ start, then $\rho' = \text{MvTsk2Top}(\rho, i)$,
      - ▹ otherwise, $\rho' = \text{RmAct}(\text{MvTsk2Top}(\rho, i), 2, 1)$,
    - ★ otherwise ($\text{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i =$ MAIN or $\text{GetRealTsk}(\rho, B) = * \wedge \text{GetTsk}(\rho, B) = S_i$),
      - ▹ if $\phi \models$ STP and $\text{TopAct}(S_i) = B$, then $b' = \neg$NOH, moreover,
        - □ if $b = \neg$NOH and $\alpha =$ start, then $\rho' = \text{MvTsk2Top}(\rho, i)$,
        - □ otherwise, $\rho' = \text{RmAct}(\text{MvTsk2Top}(\rho, i), 2, 1)$,
      - ▹ otherwise, $b' =$ NOH iff $\phi \models$ NOH, moreover,
        - □ if $b = \neg$NOH and $\alpha =$ start, then $\rho' = \text{Push}(\text{MvTsk2Top}(\rho, i), B)$,
        - □ otherwise, $\rho' = \text{RmAct}(\text{Push}(\text{MvTsk2Top}(\rho, i), B), 2, 1)$,

∗ otherwise ($i = 1$),
- · if $\phi \models$ CTK, then $\rho' = \text{ClrTsk}(\rho, B)$ and $b' =$ NOH iff $\phi \models$ NOH,
- · if $\phi \models \neg$CTK, then
  - ◇ if $\phi \models$ CTP and $B \in S_1$, then $\rho' = \text{ClrTop}(\rho, B)$, moreover,
    - ○ if $A \neq B$, then $\rho' = \text{ClrTop}(\rho, B)$, moreover,
      - ★ if $\phi \models \neg$STP, then $b' =$ NOH iff $\phi \models$ NOH,
      - ★ otherwise, $b' = \neg$NOH,
    - ○ if $A = B$,
      - ★ if $\phi \models$ STP, then
        - ▹ if $\alpha =$ start, then $\rho' = \rho$ and $b' = b$,

          - if $\alpha = \text{finishStart}$, then $\rho' = \text{RmAct}(\rho, 1, 1)$ and $b' = \neg\text{NOH}$,
        - if $\phi \models \neg\text{STP}$, then $\rho' = \rho$ and $b' = \text{NOH}$ iff $\phi \models \text{NOH}$,
    - if $\phi \models \text{CTP}$ and $B \notin S_1$, then $b' = \text{NOH}$ iff $\phi \models \text{NOH}$, moreover,
      - if $b = \neg\text{NOH}$ and $\alpha = \text{start}$, then $\rho' = \text{Push}(\rho, B)$,
      - otherwise, $\rho' = \text{RmAct}(\text{Push}(\rho, B), 1, 2)$,
    - if $\phi \models \neg\text{CTP}$, then
      - if $\phi \models \text{RTF}$ and $B \in S_1$, then
        - if $A \neq B$, then $b' = \neg\text{NOH}$, moreover,
          - if $b = \neg\text{NOH}$ and $\alpha = \text{start}$, then $\rho' = \text{MvAct2Top}(\rho, B)$,
          - otherwise, $\rho' = \text{RmAct}(\text{MvAct2Top}(\rho, B), 1, 2)$,
        - if $A = B$,
          - if $\alpha = \text{start}$, then $\rho' = \rho$ and $b' = b$,
          - if $\alpha = \text{finishStart}$, then $\rho' = \text{RmAct}(\rho, 1, 1)$ and $b' = \neg\text{NOH}$,
      - if $\phi \models \text{RTF}$ and $B \notin S_1$, then $b' = \text{NOH}$ iff $\phi \models \text{NOH}$, moreover,
        - if $b = \neg\text{NOH}$ and $\alpha = \text{start}$, then $\rho' = \text{Push}(\rho, B)$,
        - otherwise, $\rho' = \text{RmAct}(\text{Push}(\rho, B), 1, 2)$,
      - if $\phi \models \neg\text{RTF}$, then
        - if $\text{GetRealTsk}(\rho, B) = S_1$ and $\zeta_1 \neq \text{MAIN}$, then
          - if $\alpha = \text{start}$, then $\rho' = \rho$ and $b' = b$,
          - if $\alpha = \text{finishStart}$, then $\rho' = \text{RmAct}(\rho, 1, 1)$ and $b' = \neg\text{NOH}$,
        - otherwise ($\text{GetRealTsk}(\rho, B) = S_1$ and $\zeta_i = \text{MAIN}$ or $\text{GetRealTsk}(\rho, B) = * \wedge \text{GetTsk}(\rho, B) = S_1$),
          - if $\phi \models \text{STP}$ and $A = B$, or $\phi \models \text{STP} \wedge \text{PIT}$ and $\text{PreAct}(\rho) = B$,
            - if $\alpha = \text{start}$, then $\rho' = \rho$ and $b' = b$,
            - if $\alpha = \text{finishStart}$, then $\rho' = \text{RmAct}(\rho, 1, 1)$ and $b' = \neg\text{NOH}$,
          - otherwise, $b' = \text{NOH}$ iff $\phi \models \text{NOH}$, moreover,
            - if $b = \neg\text{NOH}$ and $\alpha = \text{start}$, then $\rho' = \text{Push}(\rho, B)$,
            - otherwise, $\rho' = \text{RmAct}(\text{Push}(\rho, B), 1, 2)$,
- if $\text{GetTsk}(\rho, B) = *$, then $b' = \text{NOH}$ iff $\phi \models \text{NOH}$, moreover,
  - if $b = \neg\text{NOH}$ and $\alpha = \text{start}$, then $\rho' = \text{NewTsk}(\rho, B, \text{NTK})$,
  - otherwise, $\rho' = \text{RmAct}(\text{NewTsk}(\rho, B, \text{NTK}), 2, 1)$.

Next, let us consider the subcase $\phi \models \text{TOH}$. Let ϕ' be obtained from ϕ by replacing TOH with $\neg$TOH. Moreover, suppose $\rho \xrightarrow[\tau']{\mathcal{M}} \rho'$, where $\tau' = A \xrightarrow{\alpha(\phi')} B$ and $\rho' = (\Omega_1, \cdots, \Omega_n)$.

- If $\phi \models \text{NTK} \vee \text{NDM}$, then let $\rho'' = (\Omega_1)$ and we have $\rho \xrightarrow[\tau]{\mathcal{M}} \rho''$.
- Otherwise, $\rho \xrightarrow[\tau]{\mathcal{M}} \rho'$.

C SEMANTICS OF AMASS MODELS FOR ANDROID 13.0

In this section, we present the formal semantics of AMASS models for Android 13.0, where both the activities and fragments are involved.

Fragment containers, activities, tasks, task stacks and configurations.

A *fragment container* is encoded as $V = ((F_1, n_1), \cdots, (F_k, n_k)) \in (\mathsf{Frg} \times \mathbb{N})^+$, where n_j is the identifier of an instance of F_j for $j \in [k]$ and k is called the *height* of V.

An *activity* A is encoded as a tuple (υ, η, ι), where $\upsilon = (V_1, \cdots, V_m)$ is a sequence of fragment containers associated with A, $\eta = (T_1, \cdots, T_n) \in \mathcal{CT}^*$ is the transaction stack, and ι is the assignment function that assigns to each variable x in $\mathcal{M}$ an identifier, i.e. a natural number. Note that it is possible that $m = 0$ and/or $n = 0$. For technical convenience, let ι_0 denote the assignment function that assigns each variable x the value 0.

A *task* is represented by its activity stack and is encoded as a word $S = [(A_1, \Theta_1), \cdots, (A_n, \Theta_n)]$ where $\Theta_i = (\upsilon_i, \eta_i, \iota_i)$ is a tuple of container sequence, transaction stack, and assignment function, n is called the *height* of S. The activities A_1 and A_n are called as the *top and bottom activity* of S respectively. By slight abuse of notation, for $L \subseteq \mathsf{Act}^*$, we use $S \in L$ to denote that the word $A_1 \cdots A_n$ is in L.

A *configuration* of $\mathcal{M}$ is a pair (ρ, b), where $\rho = (\Omega_1, \cdots, \Omega_n)$, and for each $i \in [n]$, $\Omega_i = (S_i, A_i, \zeta_i)$, S_i is a task, $A_i \in \mathsf{Act}$ is the real activity of S_i, $\zeta_i \in \{\mathsf{MAIN}, \mathsf{NTK}, \mathsf{NDM}, \mathsf{SIT}\}$ represents how the task S_i is launched, and $b \in \{\mathsf{NOH}, \neg\mathsf{NOH}\}$ denotes whether the topmost activity is started with NOH or not.

Let $\mathsf{Conf}_{\mathcal{M}}$ denote the set of configurations of $\mathcal{M}$. The *initial* configuration of $\mathcal{M}$ is

$$(([(A_0, ((\epsilon, \cdots, \epsilon), \epsilon, \iota_0))], A_0, \mathsf{MAIN}), \neg\mathsf{NOH}).$$

Auxiliary functions and predicates.

To specify the transition relation precisely and concisely, we define the following functions and predicates. Assume a configuration $\rho = (\Omega_1, \cdots, \Omega_n)$, and for each $i \in [n]$, $\Omega_i = (S_i, A_i, b_i)$, and a task $S = [(B_1, \Theta_1), \cdots, (B_m, \Theta_m)]$.

- $\mathsf{TopAct}(S) = B_1$, $\mathsf{BtmAct}(S) = B_m$, $\mathsf{PreAct}(S) = B_2$ if $m > 1$, and $\mathsf{PreAct}(S) = B_1$ otherwise,
- $\mathsf{TopTsk}(\rho) = S_1$, $\mathsf{TopAct}(\rho) = \mathsf{TopAct}(\mathsf{TopTsk}(\rho))$, $\mathsf{PreAct}(\rho) = \mathsf{PreAct}(\mathsf{TopTsk}(\rho))$.
- $\mathsf{Push}(\rho, B) = (([(B, ((\epsilon, \cdots, \epsilon), \epsilon, \iota_0))] \cdot S_1, A_1, \zeta_1), \Omega_2, \cdots, \Omega_n)$.
- Let $1 \le i \le n$, $S_i = [(C_1, \Theta'_1), \cdots, (C_l, \Theta'_l)]$ and $1 \le j \le l$. If $l > 1$, then

  $$\mathsf{RmAct}(\rho, i, j) = (\Omega_1, \cdots, \Omega_{i-1}, (S'_i, A_i, \zeta_i), \Omega_{i+1}, \cdots, \Omega_n),$$

  where $S'_i = [(C_1, \Theta'_1), \cdots, (C_{j-1}, \Theta'_{j-1}), (C_{j+1}, \Theta'_{j+1}), \cdots, (C_l, \Theta'_l)]$. Otherwise,

  $$\mathsf{RmAct}(\rho, i, j) = (\Omega_1, \cdots, \Omega_{i-1}, \Omega_{i+1}, \cdots, \Omega_n).$$

- $\mathsf{MvAct2Top}(\rho, B) = (([(B, \Theta)] \cdot S'_1 \cdot S''_1, A_1, \zeta_1), \Omega_2, \cdots, \Omega_n)$, if $S_1 = S'_1 \cdot [(B, \Theta)] \cdot S''_1$ with $S'_1 \in (\mathsf{Act} \setminus \{B\})^*$.
- $\mathsf{ClrTop}(\rho, B) = (([(B, \Theta)] \cdot S''_1, A_1, \zeta_1), \Omega_2, \cdots, \Omega_n)$ if $S_1 = S'_1 \cdot [B, \Theta] \cdot S''_1$ with $S'_1 \in (\mathsf{Act} \setminus \{B\})^*$. (Note that the topmost occurrence of B in S_1 is kept in $\mathsf{ClrTop}(\rho, B)$.)
- $\mathsf{ClrTop}^*(\rho, B) = (([(B, ((\epsilon, \cdots, \epsilon), \epsilon, \iota_0))] \cdot S''_1, A_1, \zeta_1), \Omega_2, \cdots, \Omega_n)$ if $S_1 = S'_1 \cdot [(B, \Theta)] \cdot S''_1$ with $S'_1 \in (\mathsf{Act} \setminus \{B\})^*$. (Note that the topmost occurrence of B in S_1 is replaced by a new B in $\mathsf{ClrTop}^*(\rho, B)$.)
- $\mathsf{ClrTsk}(\rho, B) = (([(B, ((\epsilon, \cdots, \epsilon), \epsilon, \iota_0))], A_1, \zeta_1), \Omega_2, \cdots, \Omega_n)$.
- $\mathsf{MvTsk2Top}(\rho, i) = (\Omega_i, \Omega_1, \cdots, \Omega_{i-1}, \Omega_{i+1}, \cdots, \Omega_n)$.
- $\mathsf{NewTsk}(\rho, B, \zeta) = (([(B, ((\epsilon, \cdots, \epsilon), \epsilon, \iota_0))], B, \zeta), \Omega_1, \cdots, \Omega_n)$.
- $\mathsf{GetRealTsk}(\rho, B) = S_i$ such that $i \in [n]$ is the *minimum* index satisfying $A_i = B$ if such an index i exists; $\mathsf{GetRealTsk}(\rho, B) = *$ otherwise.

- $\mathsf{GetTsk}(\rho, B) = S_i$ such that $i \in [n]$ is the *minimum* index satisfying $\mathsf{Aft}(A_i) = \mathsf{Aft}(B) \wedge \zeta_i \in \{\mathsf{NTK}, \mathsf{MAIN}\}$, if such an index i exists; $\mathsf{GetTsk}(\rho, B) = *$ otherwise.

For a container $V = ((F_1, n_1), \cdots, (F_m, n_m))$, define $\mathsf{TopFrg}(V) = F_1$.

Let $\Theta = (v, \eta, \iota)$ be the encoding of an activity A, where $\mathsf{Ctn}(A) = (i_1, \cdots, i_k)$ $(k > 0)$, $v = (V_1, \cdots, V_k)$ is the container sequence, and $\eta = (T_1, \cdots, T_l)$ $(l \geq 0)$ is the transaction stack. Moreover, let $F \in \mathsf{Frg}$, $j \in [k]$, and x be a variable. We define the following functions.

- $\mathsf{TopFrg}(v) = \{\mathsf{TopFrg}(V_i) \mid i \in [k], V_i \neq \epsilon\}$ returns the set of topmost fragments of containers in v.
- $\mathsf{ADD}(F, i_j, x)(v, \iota) = (v', \iota')$, where $v' = (V_1, \cdots, V_{j-1}, (F, n) \cdot V_j, V_{j+1}, \cdots, V_k)$, $\iota' = \iota[n/x]$, and n is the *minimum* identifier not occurring in v or ι. Intuitively, $\mathsf{ADD}(F, i_j, x)(v, \iota)$ updates (v, ι) by choosing a fresh identifier n, pushing (F, n) into the container i_j, and storing n into x.
- $\mathsf{REP}(F, i_j, x)(v, \iota) = (v', \iota')$, where $v' = (V_1, \cdots, V_{j-1}, (F, n), V_{j+1}, \cdots, V_k)$, $\iota' = \iota[n/x]$, and n is the *minimum* identifier not occurring in v or ι. Intuitively, $\mathsf{REP}(F, i_j, x)(v, \iota)$ updates v by replacing the content of container i_j with (F, n), and storing n into x.
- Suppose $V_j = ((F_1, n_1), \cdots, (F_m, n_m))$, then $\mathsf{REM}(F, i_j, x)(v, \iota) = (v', \iota')$, where
  - $v' = (V_1, \cdots, V_{j-1}, \tilde{V}, V_{j+1}, \cdots, V_k)$ such that
    - $\tilde{V} = V_j$, if $\iota(x) \neq n_{j'}$ for every $j' \in [m]$, and
    - $\tilde{V} = ((F_1, n_1), \ldots, (F_{l-1}, n_{l-1}), (F_{l+1}, n_{l+1}), \ldots, (F_m, n_m))$, if $\iota(x) = n_l$,
  - moreover, $\iota' = \iota$.

  Intuitively, the action $\mathsf{REM}(F, i_j, x)(v)$ updates v by removing the instance of F of the identifier $\iota(x)$ from container i_j and does not change ι.
- Furthermore, the functions $\mathsf{ADD}(F, i_j, n)(v, \iota)$, $\mathsf{REP}(F, i_j, n)(v, \iota)$, $\mathsf{REM}(F, i_j, n)(v, \iota)$ for concretized actions can be defined similarly (except that ι is unchanged).

For a transaction $T = (\beta_1(F_1, j_1, x_1), \cdots, \beta_r(F_r, j_r, x_r))$ such that $\beta_s \in \{\mathsf{ADD}, \mathsf{REM}, \mathsf{REP}\}$ and $F_s \in \mathsf{Frg}$ for every $s \in [r]$, $\mathsf{UpdateCtns}_T(v, \iota) = (v_r, \iota_r)$, where $(v_0, \iota_0) = (v, \iota)$, and for every $s \in [r]$, $(v_s, \iota_s) = \beta_s(F_s, j_s, x_s)(v_{s-1}, \iota_{s-1})$, i.e. $\mathsf{UpdateCtns}_T$ updates the containers and the assignment function by applying the actions in T. Furthermore, $\mathsf{UpdateCtns}_T(v, \iota)$ can be defined similarly for concretized transactions T.

We also introduce a function that concretize the actions REP by utilizing the containers in v. Suppose $F \in \mathsf{Frg}$, $j \in [k]$, x is a variable, and $V_j = ((F_1, n_1), \cdots, (F_m, n_m))$. Let n be the minimum identifier not occurring in v or ι. Then

$$\mathsf{Concretize}_{v,\iota}(\mathsf{REP}(F, i_j, x)) = \mathsf{REM}(F_1, i_j, n_1), \cdots, \mathsf{REM}(F_m, i_j, n_m), \mathsf{ADD}(F, i_j, n).$$

Moreover, let

$$\mathsf{Concretize}_{v,\iota}(\mathsf{ADD}(F, i_j, x)) = \mathsf{ADD}(F, i_j, n) \text{ and } \mathsf{Concretize}_{v,\iota}(\mathsf{REM}(F, i_j, x)) = \mathsf{REM}(F, i_j, \iota(x))$$

by convention.

Let $T = (\beta_1(F_1, j_1, x_1), \cdots, \beta_k(F_r, j_r, x_r))$ be a transaction such that $\beta_s \in \{\mathsf{ADD}, \mathsf{REM}, \mathsf{REP}\}$, and $F_s \in \mathsf{Frg}$ for every $s \in [r]$. We define $\mathsf{Concretize}_{v,\iota}(T)$ as the concatenation of the concretized action sequences $\mathsf{Concretize}_{v_0,\iota_0}(\beta_1(F_1, j_1, x_1))$, $\cdots$, $\mathsf{Concretize}_{v_{s-1},\iota_{s-1}}(\beta_s(F_s, j_s, x_s))$, where $(v_0, \iota_0) = (v, \iota)$, and for each $s \in [r]$, $(v_s, \iota_s) = \beta_s(F_s, j_s, x_s)(v_{s-1}, \iota_{s-1})$.

Finally, we define functions $\beta^{-1}(F, i_j, n)$ for $\beta(F, i_j, n) \in \mathcal{CT}$ as follows.

- $\mathsf{ADD}^{-1}(F, i_j, n)(v, \iota) = \mathsf{REM}(F, i_j, n)(v, \iota)$,
- $\mathsf{REM}^{-1}(F, i_j, n)(v, \iota) = \mathsf{ADD}(F, i_j, n)(v, \iota)$.

Intuitively, $\mathsf{ADD}(F, i_j, n)$ and $\mathsf{REM}(F, i_j, n)$ are dual actions. Moreover, for a concretized transaction

$$T = (\beta_1(F_1, j_1, n_1), \cdots, \beta_r(F_r, j_r, n_r))$$

such that $\beta_s \in \{\mathsf{ADD}, \mathsf{REM}, \mathsf{REP}\}$, $F_s \in \mathsf{Frg}$ for every $s \in [r]$, and $n \in \mathbb{N}$, we define T^{-1} as

$$(\beta_r^{-1}(F_r, j_r, n_r), \cdots, \beta_1^{-1}(F_1, j_1, n_1)).$$

Transition relation.

Let us define the semantics of $\mathcal{M}$ by a relation $(\rho, b) \xrightarrow[\tau]{\mathcal{M}} (\rho', b')$. Let (ρ, b) be the current configuration, where $\rho = ((S_1, A_1, \zeta_1), \cdots, (S_n, A_n, \zeta_n))$, for some $n \geq 1$. Let $\mathsf{Top}(\rho) = (A, \Theta)$, $\mathsf{Ctn}(A) = (i_1, \cdots, i_k)$, $\Theta = (\upsilon, \eta)$, $\upsilon = (V_1, \cdots, V_k)$, and $\eta = (T_1, \cdots, T_l)$. Suppose $S_1 = [(B_1, \Theta_1), \cdots, (B_r, \Theta_r)]$ (where $B_1 = A$ and $\Theta_1 = \Theta$).

C.1 Case $\tau = A \xrightarrow{\alpha(\phi)} B$

We first assume that $\phi \models \neg\mathsf{TOH}$. We will consider $\phi \models \mathsf{TOH}$ in the end.

$\boxed{\mathsf{Lmd}(B) = \mathsf{STD}}$

- If $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM}$ and $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, then
  - if $\phi \models \mathsf{CTP}$ and $B \in \mathsf{TopTsk}(\rho)$, then
    - if $\mathsf{TopAct}(\rho) \neq B$,
      - if $\phi \models \neg\mathsf{STP}$, $\rho' = \mathsf{ClrTop}^*(\rho, B)$ and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
      - otherwise, $\rho' = \mathsf{ClrTop}(\rho, B)$ $b' = \neg\mathsf{NOH}$,
    - if $\mathsf{TopAct}(\rho) = B$,
      - if $\phi \models \neg\mathsf{STP}$, then $\rho' = \mathsf{ClrTop}^*(\rho, B)$ and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
      - otherwise,
        - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
        - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
  - if $\phi \models \mathsf{CTP}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
    - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
    - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
  - if $\phi \models \neg\mathsf{CTP}$, then
    - if $\phi \models \mathsf{RTF}$ and $B \in \mathsf{TopTsk}(\rho)$, then
      - if $\mathsf{TopAct}(\rho) \neq B$, then $b' = \neg\mathsf{NOH}$, moreover,
        - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvAct2Top}(\rho, B)$,
        - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvAct2Top}(\rho, B), 1, 2)$,
      - if $\mathsf{TopAct}(\rho) = B$,
        - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
        - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
    - if $\phi \models \mathsf{RTF}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
      - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
      - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$.
    - if $\phi \models \neg\mathsf{RTF}$, then
      - if $\phi \models \mathsf{STP}$ and $\mathsf{TopAct}(\rho) = B$ or $\phi \models \mathsf{STP} \wedge \mathsf{PIT}$ and $\mathsf{PreAct}(\rho) = B$, then

- - - - ◇ if $\alpha = \text{start}$, then $\rho' = \rho$ and $b' = b$,
        - ◇ if $\alpha = \text{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
      - · otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
        - ◇ if $b = \neg\mathsf{NOH}$ and $\alpha = \text{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
        - ◇ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$.
- If $\phi \models \mathsf{NDM} \wedge \mathsf{MTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
  - if $b = \neg\mathsf{NOH}$ and $\alpha = \text{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NDM})$,
  - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NDM}), 2, 1)$.
- If $\phi \models \mathsf{NDM} \wedge \neg\mathsf{MTK}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$, then
    - ∗ if $i \neq 1$, then
      - · if $\phi \models \neg\mathsf{CTK}$, then
        - ◇ if $B \notin S_i$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
          - ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \text{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
          - ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
        - ◇ otherwise, $b' = \neg\mathsf{NOH}$, moreover,
          - ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \text{start}$, then $\rho' = \mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B)$,
          - ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
      - · if $\phi \models \mathsf{CTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
        - ◇ if $b = \neg\mathsf{NOH}$ and $\alpha = \text{start}$, then $\rho' = \mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B)$,
        - ◇ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
    - ∗ otherwise ($i = 1$),
      - · if $\phi \models \neg\mathsf{CTK}$, then
        - ◇ if $B \notin S_1$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
          - ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \text{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
          - ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
        - ◇ if $B \in S_1$ and $\mathsf{TopAct}(S_1) \neq B$, then $\rho' = \mathsf{ClrTop}(\rho, B)$ and $b' = \neg\mathsf{NOH}$,
        - ◇ if $B \in S_1$ and $\mathsf{TopAct}(S_1) = B$ (this implies $A = B$),
          - ∘ if $\alpha = \text{start}$, then $\rho' = \rho$ and $b' = b$,
          - ∘ if $\alpha = \text{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
      - · if $\phi \models \mathsf{CTK}$, then $\rho' = \mathsf{ClrTsk}(\rho, B)$, and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
  - if $\mathsf{GetRealTsk}(\rho, B) = *$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
    - ∗ if $b = \neg\mathsf{NOH}$ and $\alpha = \text{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NDM})$,
    - ∗ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NDM}), 2, 1)$.
- If $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \mathsf{MTK}$, or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{NDM} \wedge \mathsf{MTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
  - if $b = \neg\mathsf{NOH}$ and $\alpha = \text{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NTK})$,
  - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NTK}), 2, 1)$.
- If $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{MTK}$, or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{NDM} \wedge \neg\mathsf{MTK}$, then
  - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$, then
    - ∗ if $i \neq 1$, then
      - · if $\phi \models \mathsf{CTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,

- - - - - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B)$,
        - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
      - if $\phi \models \neg\mathsf{CTK}$, then
        - if $\phi \models \mathsf{CTP}$ and $B \in S_i$, then
          - if $\mathsf{STP} \in \phi$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
          - otherwise, $b' = \neg\mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTop}^*(\mathsf{MvTsk2Top}(\rho, i), B)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTop}^*(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
        - if $\phi \models \mathsf{CTP}$ and $B \notin S_i$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
          - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
          - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
        - if $\phi \models \neg\mathsf{CTP}$, then
          - if $\phi \models \mathsf{RTF}$ and $B \in S_i$, then $b' = \neg\mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvAct2Top}(\mathsf{MvTsk2Top}(\rho, i), B)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvAct2Top}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
          - if $\phi \models \mathsf{RTF}$ and $B \notin S_i$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
          - if $\phi \models \neg\mathsf{RTF}$, then
            - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq \mathsf{MAIN}$, then $b' = \neg\mathsf{NOH}$, moreover,
              - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
              - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
            - otherwise ($\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i = \mathsf{MAIN}$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$),
              - if $\phi \models \mathsf{STP}$ and $\mathsf{TopAct}(S_i) = B$, then $b' = \neg\mathsf{NOH}$, moreover,
                - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
                - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
              - otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
                - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
                - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
    - otherwise ($i = 1$),
      - if $\phi \models \mathsf{CTK}$, then $\rho' = \mathsf{ClrTsk}(\rho, B)$ and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
      - if $\phi \models \neg\mathsf{CTK}$, then
        - if $\phi \models \mathsf{CTP}$ and $B \in S_1$, then $\rho' = \mathsf{ClrTop}(\rho, B)$, moreover,
          - if $A \neq B$,
            - if $\phi \models \neg\mathsf{STP}$, then $\rho' = \mathsf{ClrTop}^*(\rho, B)$ and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
            - otherwise, $\rho' = \mathsf{ClrTop}(\rho, B)$ and $b' = \neg\mathsf{NOH}$,
          - if $A = B$,
            - if $\phi \models \neg\mathsf{STP}$, then $\rho' = \mathsf{ClrTop}^*(\rho, B)$ and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,

        - otherwise,
          - if α = start, then $\rho' = \rho$ and $b' = b$,
          - if α = finishStart, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
    - if $\phi \models \mathsf{CTP}$ and $B \notin S_1$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
      - if $b = \neg\mathsf{NOH}$ and α = start, then $\rho' = \mathsf{Push}(\rho, B)$,
      - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
    - if $\phi \models \neg\mathsf{CTP}$, then
      - if $\phi \models \mathsf{RTF}$ and $B \in S_1$, then
        - if $A \neq B$, then $b' = \neg\mathsf{NOH}$, moreover,
          - if $b = \neg\mathsf{NOH}$ and α = start, then $\rho' = \mathsf{MvAct2Top}(\rho, B)$,
          - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvAct2Top}(\rho, B), 1, 2)$,
        - if $A = B$,
          - if α = start, then $\rho' = \rho$ and $b' = b$,
          - if α = finishStart, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
      - if $\phi \models \mathsf{RTF}$ and $B \notin S_1$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
        - if $b = \neg\mathsf{NOH}$ and α = start, then $\rho' = \mathsf{Push}(\rho, B)$,
        - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
      - if $\phi \models \neg\mathsf{RTF}$, then
        - if $\mathsf{GetRealTsk}(\rho, B) = S_1$ and $\zeta_1 \neq \mathsf{MAIN}$,
          - if α = start, then $\rho' = \rho$ and $b' = b$,
          - if α = finishStart, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
        - otherwise ($\mathsf{GetRealTsk}(\rho, B) = S_1$ and $\zeta_i = \mathsf{MAIN}$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_1$),
          - if $\phi \models \mathsf{STP}$ and $A = B$, or $\phi \models \mathsf{STP} \wedge \mathsf{PIT}$ and $\mathsf{PreAct}(\rho) = B$,
            - if α = start, then $\rho' = \rho$ and $b' = b$,
            - if α = finishStart, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
          - otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and α = start, then $\rho' = \mathsf{Push}(\rho, B)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
- if $\mathsf{GetTsk}(\rho, B) = *$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
  - if $b = \neg\mathsf{NOH}$ and α = start, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NTK})$,
  - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NTK}), 2, 1)$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{STP}}$

- If $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM}$ and $\mathsf{Lmd}(A) \neq \mathsf{SIT}$, then
  - if $\phi \models \mathsf{CTP}$ and $B \in \mathsf{TopTsk}(\rho)$, then
    - if $\mathsf{TopAct}(\rho) \neq B$, then $\rho' = \mathsf{ClrTop}(\rho, B)$ and $b' = \neg\mathsf{NOH}$,
    - if $\mathsf{TopAct}(\rho) = B$,
      - if α = start, then $\rho' = \rho$ and $b' = b$,
      - if α = finishStart, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
  - if $\phi \models \mathsf{CTP}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,

        * if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
        * otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
    - if $\phi \models \neg\mathsf{CTP}$, then
        * if $\phi \models \mathsf{RTF}$ and $B \in \mathsf{TopTsk}(\rho)$, then
            · if $\mathsf{TopAct}(\rho) \neq B$, then $b' = \neg\mathsf{NOH}$, moreover,
                ⋄ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvAct2Top}(\rho, B)$,
                ⋄ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvAct2Top}(\rho, B), 1, 2)$,
            · if $\mathsf{TopAct}(\rho) = B$,
                ⋄ if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
                ⋄ if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
        * if $\phi \models \mathsf{RTF}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
            · if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
            · otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
        * if $\phi \models \neg\mathsf{RTF}$, then
            · if $\mathsf{TopAct}(\rho) = B$ or $\phi \models \mathsf{PIT}$ and $\mathsf{PreAct}(\rho) = B$,
                ⋄ if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
                ⋄ if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
            · otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
                ⋄ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
                ⋄ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$.
- If $\phi \models \mathsf{NDM} \wedge \mathsf{MTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
    - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NDM})$,
    - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NDM}), 2, 1)$.
- If $\phi \models \mathsf{NDM} \wedge \neg\mathsf{MTK}$, then
    - if $\mathsf{GetRealTsk}(\rho, B) = S_i$, then
        * if $i \neq 1$, then
            · if $\phi \models \neg\mathsf{CTK}$, then
                ⋄ if $B \notin S_i$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
                    ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
                    ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
                ⋄ otherwise, $b' = \neg\mathsf{NOH}$, moreover,
                    ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B)$,
                    ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
            · if $\phi \models \mathsf{CTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
                ⋄ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B)$,
                ⋄ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
        * otherwise ($i = 1$),
            · if $\phi \models \neg\mathsf{CTK}$, then
                ⋄ if $B \notin S_1$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
                    ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
                    ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,

- ⋄ if $B \in S_1$ and $\mathsf{TopAct}(S_1) \neq B$, then $\rho' = \mathsf{ClrTop}(\rho, B)$ and $b' = \neg\mathsf{NOH}$,
- ⋄ if $B \in S_1$ and $\mathsf{TopAct}(S_1) = B$ (this implies $A = B$), then
  - ∘ if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
  - ∘ if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = 0$,
- · if $\phi \models \mathsf{CTK}$, then $\rho' = \mathsf{ClrTsk}(\rho, B)$, and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
- – if $\mathsf{GetRealTsk}(\rho, B) = *$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
  - ∗ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NDM})$,
  - ∗ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NDM}), 2, 1)$.
- If $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \mathsf{MTK}$, or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{NDM} \wedge \mathsf{MTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
  - – if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NTK})$,
  - – otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NTK}), 2, 1)$.
- If $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{MTK}$, or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{NDM} \wedge \neg\mathsf{MTK}$, then
  - – if $\mathsf{GetRealTsk}(\rho, B) = S_i$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$, then
    - ∗ if $i \neq 1$, then
      - · if $\phi \models \mathsf{CTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
        - ⋄ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B)$,
        - ⋄ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
      - · if $\phi \models \neg\mathsf{CTK}$, then
        - ⋄ if $\phi \models \mathsf{CTP}$ and $B \in S_i$, then $b' = \neg\mathsf{NOH}$, moreover,
          - ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B)$,
          - ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
        - ⋄ if $\phi \models \mathsf{CTP}$ and $B \notin S_i$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
          - ∘ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
          - ∘ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
        - ⋄ if $\phi \models \neg\mathsf{CTP}$, then
          - ∘ if $\phi \models \mathsf{RTF}$ and $B \in S_i$, then $b' = \neg\mathsf{NOH}$, moreover,
            - ★ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvAct2Top}(\mathsf{MvTsk2Top}(\rho, i), B)$,
            - ★ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvAct2Top}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
          - ∘ if $\phi \models \mathsf{RTF}$ and $B \notin S_i$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
            - ★ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
            - ★ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
          - ∘ if $\phi \models \neg\mathsf{RTF}$, then
            - ★ if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq \mathsf{MAIN}$, then $b' = \neg\mathsf{NOH}$, moreover,
              - ▹ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
              - ▹ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
            - ★ otherwise ($\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i = \mathsf{MAIN}$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$),
              - ▹ if $\mathsf{TopAct}(S_i) = B$, then $b' = \neg\mathsf{NOH}$, moreover,
                - □ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
                - □ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
              - ▹ otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,

              - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
              - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
  - otherwise ($i = 1$),
    - if $\phi \models \mathsf{CTK}$, then $\rho' = \mathsf{ClrTsk}(\rho, B)$ and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
    - if $\phi \models \neg\mathsf{CTK}$, then
      - if $\phi \models \mathsf{CTP}$ and $B \in S_1$, then $\rho' = \mathsf{ClrTop}(\rho, B)$, moreover,
        - if $A \neq B$, then $\rho' = \mathsf{ClrTop}(\rho, B)$ and $b' = \neg\mathsf{NOH}$,
        - if $A = B$,
          - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
          - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
      - if $\phi \models \mathsf{CTP}$ and $B \notin S_1$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
        - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
        - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
      - if $\phi \models \neg\mathsf{CTP}$, then
        - if $\phi \models \mathsf{RTF}$ and $B \in S_1$, then
          - if $A \neq B$, then $b' = \neg\mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvAct2Top}(\rho, B)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvAct2Top}(\rho, B), 1, 2)$,
          - if $A = B$,
            - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
            - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
        - if $\phi \models \mathsf{RTF}$ and $B \notin S_1$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
          - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
          - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
        - if $\phi \models \neg\mathsf{RTF}$, then
          - if $\mathsf{GetRealTsk}(\rho, B) = S_1$ and $\zeta_1 \neq \mathsf{MAIN}$,
            - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
            - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
          - otherwise ($\mathsf{GetRealTsk}(\rho, B) = S_1$ and $\zeta_i = \mathsf{MAIN}$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_1$),
            - if $A = B$, or $\phi \models \mathsf{PIT}$ and $\mathsf{PreAct}(\rho) = B$,
              - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
              - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
            - otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
              - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
              - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
- if $\mathsf{GetTsk}(\rho, B) = *$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
  - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NTK})$,
  - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NTK}), 2, 1)$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{SIT}}$

- If $\mathsf{GetRealTsk}(\rho, B) = S_i$, then
  - if $i \neq 1$, then
    - if $\phi \models \mathsf{CTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
      - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B)$,
      - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
    - if $\phi \models \neg\mathsf{CTK}$, then $b' = \neg\mathsf{NOH}$,
      - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
      - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
  - otherwise ($i = 1$),
    - if $\phi \models \mathsf{CTK}$, then $\rho' = \mathsf{ClrTsk}(\rho, B)$ and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
    - if $\phi \models \neg\mathsf{CTK}$,
      - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
      - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
- If $\mathsf{GetRealTsk}(\rho, B) = *$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
  - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{SIT})$,
  - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{SIT}), 2, 1)$.

$\boxed{\mathsf{Lmd}(B) = \mathsf{STK}}$

- If $\mathsf{GetRealTsk}(\rho, B) = S_i$, or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$, then
  - if $i \neq 1$, then
    - if $\phi \models \mathsf{CTK}$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
      - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B)$,
      - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTsk}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
    - if $\phi \models \neg\mathsf{CTK}$ and $B \notin S_i$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover
      - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
      - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
    - if $\phi \models \neg\mathsf{CTK}$ and $B \in S_i$, then $b' = \neg\mathsf{NOH}$, moreover
      - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B)$,
      - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{ClrTop}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
  - otherwise ($i = 1$),
    - if $\phi \models \mathsf{CTK}$, then $\rho' = \mathsf{ClrTsk}(\rho, B)$ and $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$,
    - if $\phi \models \neg\mathsf{CTK}$ and $B \notin S_1$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover
      - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
      - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
    - if $\phi \models \neg\mathsf{CTK}$ and $B \in S_i$,
      - if $A \neq B$, then $\rho' = \mathsf{ClrTop}(\rho, B)$ and $b' = \neg\mathsf{NOH}$,
      - if $A = B$,
        - ⋄ if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
        - ⋄ if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
- If $\mathsf{GetTsk}(\rho, B) = *$, then $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
  - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{NewTsk}(\rho, B, \mathsf{NTK})$,

– otherwise, $\rho' = \mathsf{RmAct}(\mathsf{NewTsk}(\rho, B, \mathsf{NTK}), 2, 1)$.

At last, let us consider the situation $\phi \models \mathsf{TOH}$. Let ϕ' be obtained from ϕ by replacing TOH with $\neg$TOH. Moreover, suppose $(\rho, b) \xrightarrow[\alpha(\phi')]{\mathcal{M}} (\rho', b')$, where $\rho' = (\Omega_1, \cdots, \Omega_n)$.

- If $\phi \models \mathsf{NTK} \vee \mathsf{NDM}$ or $\mathsf{Lmd}(A) = \mathsf{SIT}$ or $\mathsf{Lmd}(B) = \mathsf{SIT}$ or $\mathsf{Lmd}(B) = \mathsf{STK}$, then let $\rho'' = (\Omega_1)$ and we have $(\rho, b) \xrightarrow[\alpha(\phi)]{\mathcal{M}} (\rho'', b')$.
- Otherwise, $(\rho, b) \xrightarrow[\mathsf{start}(\phi)]{\mathcal{M}} (\rho', b')$.

C.2 Case $\tau = A \xrightarrow{\mathsf{TS}} (\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))$

In this case, let $T = (\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))$. Then we have $(\rho, b) \xrightarrow[\tau]{\mathcal{M}} (\rho', b)$, where

$$\rho' = ((S'_1, A_1, \zeta_1), (S_2, A_2, \zeta_2), \cdots, (S_n, A_n, \zeta_n))$$

is obtained from ρ by applying all the actions in T to the containers and assignment function in $\Theta = \Theta_1$ (recall that $\mathsf{Top}(S_1) = (A, \Theta)$), specifically, $S'_1 = [(B_1, \Theta'_1), (B_2, \Theta_2), \cdots, (B_r, \Theta_r)]$, where $\Theta'_1 = (v', \eta', \iota')$ such that $(v', \iota') = \mathsf{UpdateCtns}_T(v, \iota)$ and $\eta' = \mathsf{Concretize}_{v,\iota}(T) \cdot \eta$.

C.3 Case $\tau = A \xrightarrow{\mathsf{NTS}} (\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))$

In this case, let $T = (\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))$, Then we have $(\rho, b) \xrightarrow[\tau]{\mathcal{M}} (\rho', b)$, where

$$\rho' = ((S'_1, A_1, \zeta_1), (S_2, A_2, \zeta_2), \cdots, (S_n, A_n, \zeta_n))$$

such that

$$S'_1 = [(B_1, \Theta'_1), (B_2, \Theta_2), \cdots, (B_r, \Theta_r)],$$

where $\Theta'_1 = (v', \eta, \iota')$, and $(v', \iota') = \mathsf{UpdateCtns}_T(v, \iota)$. Note that in this case, the concretization of T is not stored into the transaction stack.

C.4 Case $\tau = \mathsf{back}$

In this case, we distinguish two subcases, i.e., $\eta = \epsilon$ or $\eta \neq \epsilon$.

Case $\eta = \epsilon$

In this case, if S_1 contains exactly one activity (i.e. A), then S_1 disappears after the back action, that is, $(\rho, b) \xrightarrow[\tau]{\mathcal{M}} (\rho', \neg\mathsf{NOH})$, where $\rho' = ((S_2, A_2, \zeta_2), \cdots, (S_n, A_n, \zeta_n))$. On the other hand, if S_1 contains at least two activities, then $(\rho, b) \xrightarrow[\tau]{\mathcal{M}} (\rho', \neg\mathsf{NOH})$, where $\rho' = ((S'_1, A_1, \zeta_1), (S_2, A_2, \zeta_2), \cdots, (S_n, A_n, \zeta_n))$ and S'_1 is obtained from S_1 by removing the top activity A from S_1.

Case $\eta \neq \epsilon$

In this case, $(\rho, b) \xrightarrow[\tau]{\mathcal{M}} (\rho', b)$, where

$$\rho' = ((S'_1, A_1, \zeta_1), (S_2, A_2, \zeta_2), \cdots, (S_n, A_n, \zeta_n))$$

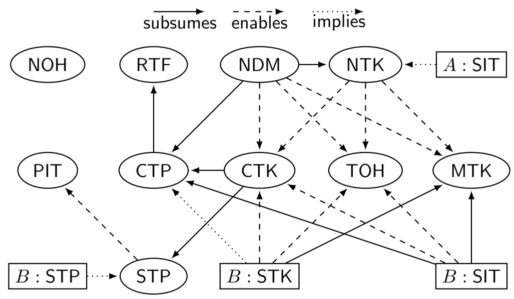

Fig. 19. Dependency graph for launch modes and intent flags in transitions $A \xrightarrow{\alpha(\phi)} B$. The launch modes (resp. the intent flags) are in boxes (resp. circles)

such that

$$S_1' = [(B_1, \Theta_1'), (B_2, \Theta_2), \cdots, (B_r, \Theta_r)],$$

where $\Theta_1' = (v', \eta', \iota')$, $(v', \iota') = \mathsf{UpdateCtns}_{T_1^{-1}}(v, \iota)$, and $\eta' = (T_2, \cdots, T_l)$. Note that T_1 is popped off the transaction stack and the actions of T_1 are revoked on (v, ι).

C.5 Other cases

When $v \neq (\epsilon, \cdots, \epsilon)$, that is, the fragment stacks of A contains at least one fragment and $F \in \mathsf{TopFrg}(v)$, then the transition rules of the following form are applicable: $F \xrightarrow{\mathsf{start}(\phi)} B$, $F \xrightarrow{\mathsf{finishStart}(\phi)} B$, $F \xrightarrow{\mathsf{TS}} (\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))$, and $F \xrightarrow{\mathsf{NTS}} (\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))$. The semantics of these rules are similar to the transition rules $A \xrightarrow{\mathsf{start}(\phi)} B$, $A \xrightarrow{\mathsf{finishStart}(\phi)} B$, $A \xrightarrow{\mathsf{TS}} (\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))$, and $A \xrightarrow{\mathsf{NTS}} (\beta_1(F_1, i_1, x_1), \cdots, \beta_k(F_k, i_k, x_k))$.

Dependencies between launch modes and intent flags

For transitions $A \xrightarrow{\alpha(\phi)} B$, the launch modes of A, B and the intent flags in ϕ may also depend on each other. The dependency can exhibit in the following three forms: n *subsumes* n', i.e., n' is ignored if n co-occurs with n', (2) n *enables* n', i.e., n' takes effect if n co-occurs with n', (3) n *implies* n', i.e., if n' subsumes (resp. enables) n'', then n subsumes (resp. enables) n'' as well. We summarize these dependencies in Figure 19, where the solid lines represent the "subsume" relation, the dashed lines represent the "enable" relation, the dotted lines represent the "implies" relation.

The following properties hold for these relations: (1) the "subsume" and "imply" relations are transitive, (2) the composition of the "imply" relation and the "subsume" (resp. "enable") relation is a subset of the "subsume" (resp. "enable") relation.

D SEMANTICS OF AMASS MODELS FOR THE OTHER VERSIONS OF ANDROID

We state the differences of the semantics of AMASS models in details. To avoid tediousness, let us focus on the situation $\phi \models \neg\mathsf{TOH}$. The differences for the situation $\phi \models \mathsf{TOH}$ are similar.

D.1 Android 11.0 and 12.0.

The semantics of AMASS for Android 11.0 and 12.0 are the same as Android 13.0.

D.2 Android 10.0, 9.0, and 8.0.

The semantics for these three versions are the same and differ from that for Android 13.0 in the following sense: RTF is ignored when used together with NTK or $\mathsf{Lmd}(A) = \mathsf{SIT}$. That is, for Android 10.0, 9.0, and 8.0, the semantics of AMASS for the case $\mathsf{Lmd}(B) = \mathsf{STD}$ and $\phi \models \mathsf{NTK} \land \neg\mathsf{NDM} \land \neg\mathsf{MTK}$, or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{NDM} \land \neg\mathsf{MTK}$, is adapted from Android 13.0 as follows.

- if $\mathsf{GetRealTsk}(\rho, B) = S_i$ or $\mathsf{GetRealTsk}(\rho, B) = * \land \mathsf{GetTsk}(\rho, B) = S_i$, then
  - if $i \neq 1$, then
    - if $\phi \models \mathsf{CTK}$, then $\cdots$
    - if $\phi \models \neg\mathsf{CTK}$, then $\cdots$
      - if $\phi \models \mathsf{CTP}$ and $B \in S_i$, then $\cdots$
      - if $\phi \models \mathsf{CTP}$ and $B \notin S_i$, then $\cdots$
      - if $\phi \models \neg\mathsf{CTP}$, then
        - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq \mathsf{MAIN}$, then $b' = \neg\mathsf{NOH}$, moreover,
          - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
          - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
        - otherwise ($\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i = \mathsf{MAIN}$ or $\mathsf{GetRealTsk}(\rho, B) = * \land \mathsf{GetTsk}(\rho, B) = S_i$),
          - if $\phi \models \mathsf{STP}$ and $\mathsf{TopAct}(S_i) = B$, then $b' = \neg\mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
          - otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
  - otherwise ($i = 1$),
    - if $\phi \models \mathsf{CTK}$, then $\cdots$,
    - if $\phi \models \neg\mathsf{CTK}$, then
      - if $\phi \models \mathsf{CTP}$ and $B \in S_1$, then $\cdots$
      - if $\phi \models \mathsf{CTP}$ and $B \notin S_1$, then $\cdots$
      - if $\phi \models \neg\mathsf{CTP}$, then
        - if $\mathsf{GetRealTsk}(\rho, B) = S_1$ and $\zeta_1 \neq \mathsf{MAIN}$,
          - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
          - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
        - otherwise ($\mathsf{GetRealTsk}(\rho, B) = S_1$ and $\zeta_i = \mathsf{MAIN}$ or $\mathsf{GetRealTsk}(\rho, B) = * \land \mathsf{GetTsk}(\rho, B) = S_1$),
          - if $\phi \models \mathsf{STP}$ and $A = B$, or $\phi \models \mathsf{STP} \land \mathsf{PIT}$ and $\mathsf{PreAct}(\rho) = B$,
            - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
            - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
          - otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,

- if $\mathsf{GetTsk}(\rho, B) = *$, then $\cdots$.

Similarly the semantics of AMASS for the case $\mathsf{Lmd}(B) = \mathsf{STP}$ and $\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{MTK}$ or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{NDM} \wedge \neg\mathsf{MTK}$ is adapted from Android 13.0 as follows.

- if $\mathsf{GetRealTsk}(\rho, B) = S_i$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$, then
  - if $i \neq 1$, then
    - if $\phi \models \mathsf{CTK}$, then $\cdots$
    - if $\phi \models \neg\mathsf{CTK}$, then $\cdots$
      - if $\phi \models \mathsf{CTP}$ and $B \in S_i$, then $\cdots$
      - if $\phi \models \mathsf{CTP}$ and $B \notin S_i$, then $\cdots$
      - if $\phi \models \neg\mathsf{CTP}$, then
        - if $\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i \neq \mathsf{MAIN}$, then $b' = \neg\mathsf{NOH}$, moreover,
          - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
          - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
        - otherwise ($\mathsf{GetRealTsk}(\rho, B) = S_i$ and $\zeta_i = \mathsf{MAIN}$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_i$),
          - if $\mathsf{TopAct}(S_i) = B$, then $b' = \neg\mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
          - otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
  - otherwise ($i = 1$),
    - if $\phi \models \mathsf{CTK}$, then $\cdots$,
    - if $\phi \models \neg\mathsf{CTK}$, then
      - if $\phi \models \mathsf{CTP}$ and $B \in S_1$, then $\cdots$
      - if $\phi \models \mathsf{CTP}$ and $B \notin S_1$, then $\cdots$
      - if $\phi \models \neg\mathsf{CTP}$, then
        - if $\mathsf{GetRealTsk}(\rho, B) = S_1$ and $\zeta_1 \neq \mathsf{MAIN}$,
          - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
          - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
        - otherwise ($\mathsf{GetRealTsk}(\rho, B) = S_1$ and $\zeta_i = \mathsf{MAIN}$ or $\mathsf{GetRealTsk}(\rho, B) = * \wedge \mathsf{GetTsk}(\rho, B) = S_1$),
          - if $A = B$, or $\phi \models \mathsf{PIT}$ and $\mathsf{PreAct}(\rho) = B$,
            - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
            - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
          - otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
            - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
            - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
- if $\mathsf{GetTsk}(\rho, B) = *$, then $\cdots$.

Note that the parts of the semantics denoted by $\cdots$ are the same as Android 13.0, and in the semantics for the situation $\phi \models \neg\mathsf{CTP}$, the flag RTF has no effects, thus is ignored.

D.3 Android 7.0.

The semantics for Android 7.0 is close to that of Android 10.0 (or 9.0, 8.0) but differs from it in the following two aspects: 1) the effect of NDM is the same as that of NTK, 2) when $\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{CTP}$, if the top task is the main task where the started activity occurs but is not the top activity, then RTF has the same effect as CTK. More precisely, for Android 7.0, only two cases "$\phi \models \mathsf{NTK}$ or $\mathsf{Lmd}(A) = \mathsf{SIT}$" and "$\phi \models \neg\mathsf{NTK}$ and $\mathsf{Lmd}(A) \neq \mathsf{SIT}$" are considered, where the semantics for the case "$\phi \models \mathsf{NTK}$ or $\mathsf{Lmd}(A) = \mathsf{SIT}$" inherits that of "$\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM}$ or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{NDM}$" for Android 10.0, while the semantics for the case "$\phi \models \neg\mathsf{NTK}$ and $\mathsf{Lmd}(A) \neq \mathsf{SIT}$" is adapted from that of "$\phi \models \neg\mathsf{NTK} \wedge \neg\mathsf{NDM}$ and $\mathsf{Lmd}(A) \neq \mathsf{SIT}$" for Android 10.0 as follows.

- If $\phi \models \mathsf{CTP}$ and $B \in \mathsf{TopTsk}(\rho)$, then $\cdots$.
- If $\phi \models \mathsf{CTP}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $\cdots$.
- If $\phi \models \neg\mathsf{CTP}$, then
  - if $\phi \models \mathsf{RTF}$ and $B \in \mathsf{TopTsk}(\rho)$, then
    - if $\mathsf{TopAct}(\rho) \neq B$, then $b' = \neg\mathsf{NOH}$, moreover,
      - if $\zeta_i = \mathsf{MAIN}$, then $\rho' = \mathsf{ClrTsk}(\rho, B)$,
      - otherwise,
        - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvAct2Top}(\rho, B)$,
        - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvAct2Top}(\rho, B), 1, 2)$,
    - if $\mathsf{TopAct}(\rho) = B$,
      - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
      - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
  - if $\phi \models \mathsf{RTF}$ and $B \notin \mathsf{TopTsk}(\rho)$, then $\cdots$,
  - if $\phi \models \neg\mathsf{RTF}$, then $\cdots$.

D.4 Android 6.0.

The semantics for Android 6.0 differs from that of Android 10.0 (or 9.0, 8.0) in the following two aspects: 1) the effect of NDM is the same as that of NTK, 2) the task allocation mechanism of Android 6.0 does not use the real activities of tasks and only relies on affinities. More precisely, for Android 6.0, only two cases "$\phi \models \mathsf{NTK}$ or $\mathsf{Lmd}(A) = \mathsf{SIT}$" and "$\phi \models \neg\mathsf{NTK}$ and $\mathsf{Lmd}(A) \neq \mathsf{SIT}$" are considered, where the semantics for the case "$\phi \models \mathsf{NTK}$ or $\mathsf{Lmd}(A) = \mathsf{SIT}$" inherits that of "$\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM}$ or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{NDM}$" for Android 10.0, while the semantics for the case "$\phi \models \mathsf{NTK}$ or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{MTK}$" is adapted from that of "$\phi \models \mathsf{NTK} \wedge \neg\mathsf{NDM} \wedge \neg\mathsf{MTK}$ or $\mathsf{Lmd}(A) = \mathsf{SIT}$ and $\phi \models \neg\mathsf{NDM} \wedge \mathsf{MTK}$" for the case $\mathsf{Lmd}(B) = \mathsf{STD}$ for Android 10.0 as follows, where the conditions involving $\mathsf{GetRealTsk}(\rho, B)$ and $\mathsf{GetTsk}(\rho, B)$ are simplified into the conditions involving only $\mathsf{GetTsk}(\rho, B)$, moreover, we do not need to distinguish whether a task is the main task or not.

- if $\mathsf{GetTsk}(\rho, B) = S_i$, then
  - if $i \neq 1$, then
    - if $\phi \models \mathsf{CTK}$, then $\cdots$
    - if $\phi \models \neg\mathsf{CTK}$, then $\cdots$

      - if $\phi \models \mathsf{CTP}$ and $B \in S_i$, then $\cdots$
      - if $\phi \models \mathsf{CTP}$ and $B \notin S_i$, then $\cdots$
      - if $\phi \models \neg\mathsf{CTP}$, then
        - if $\phi \models \mathsf{STP}$ and $\mathsf{TopAct}(S_i) = B$, then $b' = \neg\mathsf{NOH}$, moreover,
          - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
          - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
        - otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
          - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
          - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
  - otherwise ($i = 1$),
    - if $\phi \models \mathsf{CTK}$, then $\cdots$,
    - if $\phi \models \neg\mathsf{CTK}$, then
      - if $\phi \models \mathsf{CTP}$ and $B \in S_1$, then $\cdots$
      - if $\phi \models \mathsf{CTP}$ and $B \notin S_1$, then $\cdots$
      - if $\phi \models \neg\mathsf{CTP}$, then
        - if $\phi \models \mathsf{STP}$ and $A = B$, or $\phi \models \mathsf{STP} \wedge \mathsf{PIT}$ and $\mathsf{PreAct}(\rho) = B$,
          - if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
          - if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
        - otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
          - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
          - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
- if $\mathsf{GetTsk}(\rho, B) = *$, then $\cdots$.

Similarly for the case $\mathsf{Lmd}(B) = \mathsf{STP}$, the semantics is adapted as follows:

- if $\mathsf{GetTsk}(\rho, B) = S_i$, then
  - if $i \neq 1$, then
    - if $\phi \models \mathsf{CTK}$, then $\cdots$
    - if $\phi \models \neg\mathsf{CTK}$, then $\cdots$
      - if $\phi \models \mathsf{CTP}$ and $B \in S_i$, then $\cdots$
      - if $\phi \models \mathsf{CTP}$ and $B \notin S_i$, then $\cdots$
      - if $\phi \models \neg\mathsf{CTP}$, then
        - if $\mathsf{TopAct}(S_i) = B$, then $b' = \neg\mathsf{NOH}$, moreover,
          - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{MvTsk2Top}(\rho, i)$,
          - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{MvTsk2Top}(\rho, i), 2, 1)$,
        - otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
          - if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B)$,
          - otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\mathsf{MvTsk2Top}(\rho, i), B), 2, 1)$,
  - otherwise ($i = 1$),
    - if $\phi \models \mathsf{CTK}$, then $\cdots$,
    - if $\phi \models \neg\mathsf{CTK}$, then
      - if $\phi \models \mathsf{CTP}$ and $B \in S_1$, then $\cdots$

- $\cdot$ if $\phi \models \mathsf{CTP}$ and $B \notin S_1$, then $\cdots$
- $\cdot$ if $\phi \models \neg\mathsf{CTP}$, then
  - $\diamond$ if and $A = B$, or $\phi \models \mathsf{PIT}$ and $\mathsf{PreAct}(\rho) = B$,
    - $\circ$ if $\alpha = \mathsf{start}$, then $\rho' = \rho$ and $b' = b$,
    - $\circ$ if $\alpha = \mathsf{finishStart}$, then $\rho' = \mathsf{RmAct}(\rho, 1, 1)$ and $b' = \neg\mathsf{NOH}$,
  - $\diamond$ otherwise, $b' = \mathsf{NOH}$ iff $\phi \models \mathsf{NOH}$, moreover,
    - $\circ$ if $b = \neg\mathsf{NOH}$ and $\alpha = \mathsf{start}$, then $\rho' = \mathsf{Push}(\rho, B)$,
    - $\circ$ otherwise, $\rho' = \mathsf{RmAct}(\mathsf{Push}(\rho, B), 1, 2)$,
- if $\mathsf{GetTsk}(\rho, B) = *$, then $\cdots$.

Similarly for the case $\mathsf{Lmd}(B) = \mathsf{STK}$, the semantics is adapted as follows where the conditions involving $\mathsf{GetRealTsk}(\rho, B)$ and $\mathsf{GetTsk}(\rho, B)$ are simplified into the conditions involving only $\mathsf{GetTsk}(\rho, B)$:

- If $\mathsf{GetTsk}(\rho, B) = S_i$, then $\cdots$
- If $\mathsf{GetTsk}(\rho, B) = *$, then $\cdots$.

E AUDITING THE SOURCE CODE OF ANDROID OS

In this section, we audit the source code of Android OS for AMASS_{ACT} and AMASS_{FRG}.

E.1 Auditing the source code for AMASS_{ACT}

We examine the source code of the procedures called directly or indirectly by startActivityInner().

- From the source code in line 2550 of the file activityStarter.java (see Figure 20), the procedure computeLaunchingTaskFlags() computes the intent flags implied by the launch modes of the starting and started activities.
- From the source code in line 2678 of the file activityStarter.java (see Figure 21), the procedure getReusableTask() calls findTask() to compute a reusable task to put the started activity, when either the intent flag FLAG_ACTIVITY_NEW_TASK is set to true and the flag FLAG_ACTIVITY_MULTIPLE_TASK is set to false, or the launch modes of the started activity is singleInstance or singleTask.
  - In line 2301 of the file RootWindowContainer.java (see Figure 21), findTask() calls process(), which then calls forAllLeafTasks(this) in line 338 of the file RootWindowContainer.java, and in the procedure forAllLeafTasks(), test(this) is called in line 3221 of the file Task.java.
  - In line 394 of the file RootWindowContainer.java (see Figure 21), the procedure test(Task *task*) first checks whether the real activity of *task* matches the started activity. Otherwise, in line 401, it checks whether the affinity of *task* matches the task affinity of the started activity. Therefore, from the source code of the procedure test(), we confirm that *the task allocation mechanism defined in the semantics of* AMASS_{ACT} *in Section 4.1.1 matches its actual implementation in Android OS.*
- When getReusableTask() does not find a task, computeTargetTask() (see line 1880 of Figure 21) will be called to see whether the top task in the task stack can be used to put the started activity. If the answer is yes, then computeTargetTask() returns the top task.
- If either getReusableTask() or computeTargetTask() finds an existing task to put the started activity, then recycleTask() (see line 2037 of of Figure 22) is called to prepare the task to be reused for this launch, where complyActivityFlags() is called to comply with the specified intent flags.

```
// +/master/services/core/java/com/android/server/wm/ActivityStarter.java
1631   int startActivityInner(final ActivityRecord r, ActivityRecord sourceRecord,
1632       IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
1633       int startFlags, ActivityOptions options, Task inTask,
1634       TaskFragment inTaskFragment, @BalCode int balCode,
1635       NeededUriGrants intentGrants, int realCallingUid) {
           ...
1639       computeLaunchingTaskFlags();
           ...
1655       final Task reusedTask = getReusableTask();
           ...
1666       final Task targetTask = reusedTask != null ? reusedTask : computeTargetTask();
1667       final boolean newTask = targetTask == null;
           ...
1715       startResult =
1716           recycleTask(targetTask, targetTaskTop, reusedTask, intentGrants);
           ...
1738       if (newTask) {
1739           final Task taskToAffiliate = (mLaunchTaskBehind && mSourceRecord != null)
1740                   ? mSourceRecord.getTask() : null;
1741           setNewTask(taskToAffiliate);
1742       } else if (mAddingToTask) {
1743           addOrReparentStartingActivity(targetTask, "addingtotask");
1744       }
           ...
1842   }

// +/master/services/core/java/com/android/server/wm/ActivityStarter.java
2550   private void computeLaunchingTaskFlags() {
           ...
2619       } else if (mSourceRecord.launchMode == LAUNCH_SINGLE_INSTANCE) {
               ...
2623           mLaunchFlags |= FLAG_ACTIVITY_NEW_TASK;
2624       } else if (isLaunchModeOneOf(LAUNCH_SINGLE_INSTANCE, LAUNCH_SINGLE_TASK)) {
               ...
2627           mLaunchFlags |= FLAG_ACTIVITY_NEW_TASK;
2628       }
           ...
2636   }
```

Fig. 20. Source code of ActivityStarter.startActivityInner() and ActivityStarter.computeLaunchingTaskFlags()

From the source code of complyActivityFlags() (see line 2182 of ActivityStarter.java in Figure 21), we can see that it deals with the intent flags in the following way.

- In line 2191, if the intent flags FLAG_ACTIVITY_NEW_TASK and FLAG_ACTIVITY_CLEAR_TASK are both set to true, then complyActivityFlags() calls the procedure performClearTaskForReuse() in line 2199 to clear all activities in the task, and sets the Boolean variable $mAddingToTask$ to true in line 2201, so that the started activity will be added into the target task when the procedure addOrReparentStartingActivity() is called.
- Otherwise, that is, FLAG_ACTIVITY_NEW_TASK or FLAG_ACTIVITY_CLEAR_TASK is set to false, then complyActivityFlags() performs the following operations.
    - ∗ In line 2203, if FLAG_ACTIVITY_CLEAR_TOP is set to true, then complyActivityFlags() calls the procedure performClearTop() in line 2211 to clear all the activities on top of the started activity.
    - ∗ In line 2245, if FLAG_ACTIVITY_CLEAR_TOP is set to false and FLAG_ACTIVITY_REORDER_TO_FRONT is set to true, then complyActivityFlags() calls the procedure moveActivityToFront() in line 2255 to move the started activity to the top of the task.

```
// +/master/services/core/java/com/android/server/wm/ActivityStarter.java
2642    private Task getReusableTask() {
            ...
2657        boolean putIntoExistingTask = ((mLaunchFlags & FLAG_ACTIVITY_NEW_TASK) != 0 &&
2658                (mLaunchFlags & FLAG_ACTIVITY_MULTIPLE_TASK) == 0)
2659                || isLaunchModeOneOf(LAUNCH_SINGLE_INSTANCE, LAUNCH_SINGLE_TASK);
            ...
2665        if (putIntoExistingTask) {
                ...
2678            intentActivity =
2679                    mRootWindowContainer.findTask(mStartActivity, mPreferredTaskDisplayArea);
                ...
2681        }
2699    }

// +/master/services/core/java/com/android/server/wm/ActivityStarter.java
1880    private Task computeTargetTask() {
            ...
1898        final ActivityRecord top = rootTask.getTopNonFinishingActivity();
            ...
1900        return top.getTask();
            ...
1907    }

// +/master/services/core/java/com/android/server/wm/RootWindowContainer.java
2301    ActivityRecord findTask(int activityType, String taskAffinity, Intent intent, ActivityInfo info,
2302            TaskDisplayArea preferredTaskDisplayArea) {
            ...
2324        mTmpFindTaskResult.process(taskDisplayArea);
            ...
2338    }

// +/master/services/core/java/com/android/server/wm/RootWindowContainer.java
326     void process(WindowContainer parent) {
            ...
338         parent.forAllLeafTasks(this);
339     }

// +/master/services/core/java/com/android/server/wm/Task.java
3209    boolean forAllLeafTasks(Predicate<Task> callback) {
            ...
3221        return callback.test(this);
            ...
3224    }

// +/master/services/core/java/com/android/server/wm/RootWindowContainer.java
342     public boolean test(Task task) {
            ...
394         if (task.realActivity != null && task.realActivity.compareTo(cls) == 0
395                 && Objects.equals(documentData, taskDocumentData)) {
                ...
400             return true;
401         } else if (affinityIntent != null && affinityIntent.getComponent() != null
402                 && affinityIntent.getComponent().compareTo(cls) == 0 &&
403                 Objects.equals(documentData, taskDocumentData)) {
                ...
407             return true;
408         }
            ...
425     }
```

Fig. 21. Source code of ActivityStarter.getReusableTask(), RootWindowContainer.findTask(), RootWindowContainer.process(), Task.forAllLeafTasks(), and RootWindowContainer.test()

    - In line 2271, if FLAG_ACTIVITY_CLEAR_TOP and FLAG_ACTIVITY_REORDER_TO_FRONT are both set to false, moreover, the real activity of the target task is the same as the started activity, then the content of the target task will be not changed.
    - In line 2275, if FLAG_ACTIVITY_CLEAR_TOP and FLAG_ACTIVITY_REORDER_TO_FRONT are both set to false, and FLAG_ACTIVITY_SINGLE_TOP is set to true, moreover, the top activity of the target task is the same as the started activity, then the content of the target task will not be changed.
    - In line 2291, if none of the aforementioned situations happens, then the Boolean variable $mAddingToTask$ is set to true in line 2292, so that the started activity will be added into the target task when the procedure addOrReparentStartingActivity() is called.

  Therefore, we confirm that *the way of dealing with the intent flags in the semantics of AMASS$_{ACT}$ in Section 4.1.1 is consistent with source code of complyActivityFlags().*
- If neither getReusableTask() nor computeTargetTask() finds an existing task, then setNewTask() (see line 2842 of the file ActivityStarter.java in Figure 22) is called to start a new task. Moreover, in line 2844, setNewTask() calls reuseOrCreateTask() to create a new task. From the source code of reuseOrCreateTask() as illustrated in Figure 23, the procedure reuseOrCreateTask() creates a new task in line 5959. Note that since the condition canReuseAsLeafTask() matters only when multi-screen mode is enabled, we ignore it in this work. As a result, in the single-screen mode, reuseOrCreateTask() always creates a new task.
- From the source code of the procedure addOrReparentStartingActivity() in Figure 22, the started activity is added to the target task by calling addChild() in line 2910.

In summary, after auditing the Android source code for starting an activity, we confirm that the semantics of AMASS$_{ACT}$ is consistent with its actual implementation in Android OS. In particular, the task allocation mechanism and the intent flags in the semantics conform to the source code in Android OS.

E.2 Auditing the source code for AMASS$_{FRG}$

From the source code in Figure 24, the procedure executeOpsTogether() executes multiple fragment transactions stored in *records* together. At first, for each *record* in *records*, it either calls expandOps() or trackAddedFragmentsInPop() to update the fragments in *added* and transforms each “replace” action of *record* into a sequence of “add” and “remove” actions. The actual executions of these fragment transactions are fulfilled by calling executeOps() in line 2205.

The procedure expandOps() expands the actions in a fragment transaction by transforming each “replace” action into a sequence of “add” and “remove” actions. From the source code of expandOps() (cf. Figure 24), when op with $op.cmd$ = OP_REPLACE is processed, a sequence of “remove” actions followed by an “add” action is added to mOps as follows. For each fragment in the current fragment container (that is, the fragment in *added* such that $old.mContainerId$ == $containerId$), a “remove” action is added (line 932). Finally, $op.cmd$ is changed from OP_REPLACE to OP_ADD (line 942). Therefore, we confirm that *the way of dealing with REP actions in the semantics of AMASS$_{FRG}$ in Section 4.1.2 is consistent with the source code of expandOps().*

When committing a fragment transaction, FragmentManager.executeOps() calls BackStackRecord.executeOps(). From the source code of BackStackRecord.executeOps() in Figure 25 (line 759-809), each action in *mOps* is executed by adding a fragment to or removing a fragment from the corresponding fragment container. Evidently, *the execution of fragment actions as defined in the semantics of AMASS$_{FRG}$ in Section 4.1.2 is consistent with the source code of BackStackRecord.executeOps().*

```
// +/master/services/core/java/com/android/server/wm/ActivityStarter.java
2037   int recycleTask(Task targetTask, ActivityRecord targetTaskTop, Task reusedTask,
2038         NeededUriGrants intentGrants, BalVerdict balVerdict) {
            ...
2090       complyActivityFlags(targetTask,
2091             reusedTask != null ? reusedTask.getTopNonFinishingActivity() : null, intentGrants);
            ...
2127   }

// +/master/services/core/java/com/android/server/wm/ActivityStarter.java
2182   private void complyActivityFlags(Task targetTask, ActivityRecord reusedActivity,
2183         NeededUriGrants intentGrants) {
            ...
2191       if ((mLaunchFlags & (FLAG_ACTIVITY_NEW_TASK | FLAG_ACTIVITY_CLEAR_TASK))
2192             == (FLAG_ACTIVITY_NEW_TASK | FLAG_ACTIVITY_CLEAR_TASK)) {
                ...
2199          targetTask.performClearTaskForReuse(true /* excludingTaskOverlay*/);
                ...
2201           mAddingToTask = true;
                ...
2203       } else if ((mLaunchFlags & FLAG_ACTIVITY_CLEAR_TOP) != 0
                    ...
2206                   LAUNCH_SINGLE_INSTANCE_PER_TASK)) {
                ...
2211           final ActivityRecord clearTop = targetTask.performClearTop(mStartActivity,
                ...
2245       } else if ((mLaunchFlags & FLAG_ACTIVITY_CLEAR_TOP) == 0 && !mAddingToTask
2246             && (mLaunchFlags & FLAG_ACTIVITY_REORDER_TO_FRONT) != 0) {
                ...
2253           if (act != null) {
2254               final Task task = act.getTask();
2255               boolean actuallyMoved = task.moveActivityToFront(act);
                    ...
2270           }
2271       } else if (mStartActivity.mActivityComponent.equals(targetTask.realActivity)) {
2272           if (targetTask == mInTask) {
                ...
2275       } else if (((mLaunchFlags & FLAG_ACTIVITY_SINGLE_TOP) != 0
2276                       || LAUNCH_SINGLE_TOP == mLaunchMode)
2277           && targetTaskTop.mActivityComponent.equals(mStartActivity.mActivityComponent)
2278           && mStartActivity.resultTo == null) {
                    ...
2291           } else if (reusedActivity == null) {
2292               mAddingToTask = true;
2293           }
                ...
2307       }
2308   }

// +/master/services/core/java/com/android/server/wm/ActivityStarter.java
2842   private void setNewTask(Task taskToAffiliate) {
            ...
2844       final Task task = mTargetRootTask.reuseOrCreateTask(
            ...
2856   }

// +/master/services/core/java/com/android/server/wm/ActivityStarter.java
2870   private void addOrReparentStartingActivity(@NonNull Task task, String reason) {
2871       TaskFragment newParent = task;
            ...
2910       newParent.addChild(mStartActivity, POSITION_TOP);
            ...
2914   }
```

Fig. 22. Source code of ActivityStarter.recycleTask(), ActivityStarter.complyActivityFlags(), ActivityStarter.setNewTask(), ActivityStarter.addOrReparentStartingActivity()

```
// +/master/services/core/java/com/android/server/wm/Task.java
    Task reuseOrCreateTask(ActivityInfo info, Intent intent, IVoiceInteractionSession voiceSession,
            IVoiceInteractor voiceInteractor, boolean toTop, ActivityRecord activity,
            ActivityRecord source, ActivityOptions options) {
        ...
        if (canReuseAsLeafTask()) {
            ...
        } else {
            ...
            task = new Task.Builder(mAtmService)
            ...
                    .build();
        }
        ...
    }
```

Fig. 23. Source code of Task.reuseOrCreateTask()

When popping a fragment transaction, executeOpsTogether() calls trackAddedFragmentsInPop(). From the source code of trackAddedFragmentsInPop() in Figure 24, the fragments in *added* are updated by revoking all "add" or "remove" actions in *mOps* (i.e. all the actions in the current fragment transaction stack). Note that *added* is a temporary data structure used for expanding "replace" actions and is different from the fragment stacks. The fragment stacks are updated by calling BackStackRecord.executePopOps() in Figure 25, where for each *op* in *mOPs*, if $op.cmd$=OP_ADD (resp. OP_REMOVE), $op.fragment$ is removed from *mManager* (resp. added to *mManager*). Therefore, we confirm that *the way of revoking fragment transactions in the semantics of AMASS*$_{FRG}$ *in Section 4.1.2 is consistent with the source code of trackAddedFragmentsInPop().*

F VALIDATION OF THE SEMANTICS OF AMASS MODELS

We use ValApp to validate the semantics of AMASS models. In the sequel, for each version of Android, we validate the semantics of AMASS models for this version. We utilize the automated methods as described in Section **??** to validate the semantics.

Recall that the transition rules of AMASS models are of the following three forms:

- $\tau = A \xrightarrow{\alpha(\phi)} B$, $\tau = F \xrightarrow{\alpha(\phi)} B$,
- $\tau = A \xrightarrow{\mu} T$, $\tau = F \xrightarrow{\mu} T$,
- $\tau =$ back.

Validate the semantics of the transition rules $\tau = A \xrightarrow{\alpha(\phi)} B$ *or* $\tau = F \xrightarrow{\alpha(\phi)} B$. We illustrate the validation of the semantics for $\tau = A \xrightarrow{\alpha(\phi)} B$. The validation of the semantics for $\tau = F \xrightarrow{\alpha(\phi)} B$ is similar. To valid the semantics of $\tau = A \xrightarrow{\alpha(\phi)} B$, we generate one configuration for each combination of the launch modes of A and B, the values of α, the intent flags, and the constraints on the configuration before applying the transition, e.g. $\text{GetRealTsk}(\rho, B) = *$, $\text{GetTsk}(\rho, B) = S_1$, and $B \in \text{TopTsk}(\rho)$. In total, there are $901,120$ different combinations to be considered and we generate configurations for all of them. Then we use these configurations to validate the formal semantics. Through experiments, we discover that for every combination, the configuration obtained by applying the transition rule corresponding to the combination according to the formal semantics and the actual configuration returned by ADB are equal, thus the formal semantics of AMASS in this case for Android 13.0 are confirmed to be consistent with the actual behaviors of Android apps.

```
// +/master/core/java/android/app/FragmentManager.java
2178    private void executeOpsTogether(ArrayList<BackStackRecord> records,
2179            ArrayList<Boolean> isRecordPop, int startIndex, int endIndex) {
        ...
2189    for (int recordNum = startIndex; recordNum < endIndex; recordNum++) {
2190        final BackStackRecord record = records.get(recordNum);
2191        final boolean isPop = isRecordPop.get(recordNum);
2192        if (!isPop) {
2193            oldPrimaryNav = record.expandOps(mTmpAddedFragments, oldPrimaryNav);
2194        } else {
2195            record.trackAddedFragmentsInPop(mTmpAddedFragments);
2196        }
            ...
2205        executeOps(records, isRecordPop, startIndex, endIndex);
            ...
2236    }

// +/master/core/java/android/app/BackStackRecord.java
892     Fragment expandOps(ArrayList<Fragment> added, Fragment oldPrimaryNav) {
893         for (int opNum = 0; opNum < mOps.size(); opNum++) {
894             final Op op = mOps.get(opNum);
895             switch (op.cmd) {
                    ...
910                 case OP_REPLACE: {
911                     final Fragment f = op.fragment;
                        ...
914                     for (int i = added.size() - 1; i >= 0; i--) {
915                         final Fragment old = added.get(i);
916                         if (old.mContainerId == containerId) {
                                ...
927                             final Op removeOp = new Op(OP_REMOVE, old);
                                ...
932                             mOps.add(opNum, removeOp);
933                             added.remove(old);
                                ...
937                     }
                        ...
942                     op.cmd = OP_ADD;
943                     added.add(f);
            ...
959     }

// +/master/core/java/android/app/BackStackRecord.java
968     void trackAddedFragmentsInPop(ArrayList<Fragment> added) {
969         for (int opNum = 0; opNum < mOps.size(); opNum++) {
970             final Op op = mOps.get(opNum);
971             switch (op.cmd) {
972                 case OP_ADD:
973                 case OP_ATTACH:
974                     added.remove(op.fragment);
975                     break;
976                 case OP_REMOVE:
977                 case OP_DETACH:
978                     added.add(op.fragment);
979                     break;
980             }
982         }
982     }
```

Fig. 24. Source code of executeOpsTogether(), expandOps(), and trackAddedFragmentsInPop()

Validation of the semantics of the transition rules $\tau = A \xrightarrow{\mu} T$ *or* $F \xrightarrow{\mu} T$. We illustrate the validation of the semantics for $\tau = A \xrightarrow{\mu} T$. The validation of the semantics for $F \xrightarrow{\mu} T$ is similar. Note that according to the definition of semantics of AMASS models in Appendix C, the only requirement for the enablement of $A \xrightarrow{\mu} T$ is that the top activity is A. This

```
// +/master/core/java/android/app/FragmentManager.java
2397   private static void executeOps(ArrayList<BackStackRecord> records,
2398         ArrayList<Boolean> isRecordPop, int startIndex, int endIndex) {
           ...
2402       if (isPop) {
               ...
2407           record.executePopOps(moveToState);
2408       } else {
               ...
2410           record.executeOps();
2411       }
           ...
2413   }

// +/master/core/java/android/app/BackStackRecord.java
759    void executeOps() {
760        final int numOps = mOps.size();
761        for (int opNum = 0; opNum < numOps; opNum++) {
               ...
767            switch (op.cmd) {
768                case OP_ADD:
769                    f.setNextAnim(op.enterAnim);
770                    mManager.addFragment(f, false);
771                    break;
772                case OP_REMOVE:
773                    f.setNextAnim(op.exitAnim);
774                    mManager.removeFragment(f);
775                    break;
               ...
800            }
          ...
809    }

// +/master/core/java/android/app/BackStackRecord.java
818    void executePopOps(boolean moveToState) {
819        for (int opNum = mOps.size() - 1; opNum >= 0; opNum--) {
820        final Op op = mOps.get(opNum);
821        Fragment f = op.fragment;
               ...
826         switch (op.cmd) {
827             case OP_ADD:
828                 f.setNextAnim(op.popExitAnim);
829                 mManager.removeFragment(f);
830                 break;
831             case OP_REMOVE:
832                 f.setNextAnim(op.popEnterAnim);
833                 mManager.addFragment(f, false);
834                 break;
               ...
859            }
          ...
867    }
```

Fig. 25. Source code of FragmentManager.executeOps(), BackStackRecord.executeOps(), and BackStackRecord.executePopOps()

requirement does not constrain the source configurations very much. To validate the semantics of $\tau = A \xrightarrow{\mu} T$, we fix the values of the following parameters and generate configurations as well as transition rules with these values.

- The number of containers associated with A is 1 for each $A \in \mathrm{Act}$,
- the maximum number of fragment transactions in the transaction stack is 1,
- the maximum number of actions in a fragment transaction is 2,
- the identifiers in (concretized) actions in a fragment transaction are from the set $\{1, 2\}$.

Note that the maximum number of fragments in a container is bounded by $\hbar = 6$ (i.e. the bound on the height of the stacks). In total, there are $4,032$ different configurations to be considered. In the end, we generate $4,032$ configurations for ValApp. We also generate 576 (288 if only $\tau = A \xrightarrow{\mu} T$ is counted) transition rules for ValApp. Therefore, the total number of (configuration, transition rule) pairs for ValApp is $1,741,824$ ($1,161,216$ if $\tau = A \xrightarrow{\mu} T$ is counted). Then for all these pairs, we apply the transition rules on the configurations to validate the formal semantics. Through experiments, we discover that for each configuration and each transition rule, the configuration obtained by applying the transition rule according to the formal semantics and the actual configuration returned by ADB are equal, thus the formal semantics of AMASS models in this case for Android 13.0 are confirmed to be consistent with the actual behaviors of Android apps.

Validation of the semantics of the transition rule $\tau =$ back. According to the definition of semantics of AMASS models in Appendix C, there are only two constraints on the configurations, i.e., $\eta = \epsilon$ and $\eta \neq \epsilon$. Because the aforementioned 4,032 configurations generated for $\tau = A \xrightarrow{\mu} T$ or $F \xrightarrow{\mu} T$ cover both constraints $\eta = \epsilon$ and $\eta \neq \epsilon$. Hence we could reuse these $4,032$ configurations for the validation of the semantics for $\tau =$ back. Then for all these configurations, we apply the transition rule $\tau =$ back on the configurations to validate the formal semantics. Through experiments, we discover that for each configuration and each transition rule, the configuration obtained by applying the transition rule according to the formal semantics and the actual configuration returned by ADB are equal, thus the formal semantics of AMASS models in this case for Android 13.0 are confirmed to be consistent with the actual behaviors of Android apps.

All the results of the validation experiments for AMASS models are available at https://github.com/Jinlong-He/TaskDroid/blob/master/semanticsVal.md.

G INFORMATION ABOUT RANDOMLY SELECTED TASK/FRAGMENT-CONTAINER UNBOUNDED APPS

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

Package name	|Act|	|Frg|	$\|\Delta\|$	**Size(MB)**
org.gnucash.android	10	2	32	2.71
systems.byteswap.aiproute	3	0	3	1.08
max.music_cyclon	2	0	2	1.81
com.matburt.mobileorg	8	0	9	1.36
org.npr.android.news	5	0	8	0.92
com.commonsware.android.arXiv	8	0	12	0.52
net.mabako.steamgifts	6	1	24	2.66
com.app.Zensuren	17	0	22	0.17
uk.co.busydoingnothing.prevo	4	0	5	11.79
de.drhoffmannsoftware.calcvac	14	0	24	0.9
org.sasehash.burgerwp	2	0	3	1.76
com.mikifus.padland	6	1	18	1.07
org.evilsoft.pathfinder.reference	5	0	8	22.02
com.android.keepass	4	0	6	1.65
de.bloosberg.basti.childresuscalc	2	0	2	1.41
com.samebits.beacon.locator	2	0	2	1.98
ohm.quickdice	9	0	14	1.41
org.saiditnet.redreader	11	1	51	4.99
com.gianlu.aria2app	13	0	31	21.11
net.artificialworlds.rabbitescape	3	0	3	18.81
com.sam.hex	10	0	23	0.76
com.kaliturin.blacklist	2	0	2	2.06
de.schildbach.oeffi	10	0	35	1.79
com.aurora.store	22	0	56	5.47
com.adrienpoupa.attestationcoronavirus	2	0	2	3.58

Table 20. Statistics of 25 randomly selected “task-unbounded” apps from F-Droid

Package name	|Act|	|Frg|	|Δ|	Size(MB)
com.abdulqawi.ali.mosabqa	8	0	25	2.84
com.music.star.player	3	3	25	2.38
com.drclabs.android.wootchecker	11	0	18	0.84
com.hotels.hotelsmecca	6	1	16	16.43
com.airg.hookt	6	0	12	10.59
com.holidu.holidu	2	0	2	4.04
com.appkey.english3000freekata	28	0	97	1.06
socials.com.application	34	0	122	3.75
www.genting.rwgenting	3	0	5	12.78
com.goldenhammer.beisboldominicana	3	1	10	4.58
com.travolution.seoultravelpass	5	0	10	4.58
com.ic.myMoneyTracker	25	0	60	3.39
tekcarem.gebeliktakibi	59	0	255	3.15
de.fckoeln.app	22	0	45	40.77
de.twokit.castbrowser	5	0	8	3.79
com.TWTD.FLIXMOVIE	6	0	10	7.73
com.emeint.android.mwallet.tub	10	0	95	59.02
com.ctt.celltrak2	22	0	80	11.31
biz.mobinex.android.apps.cep_sifrematik	11	0	34	3.54
com.linesmarts.linesmartsfree	4	0	4	28.47
com.nhiApp.v1	8	9	107	26.6
com.alfarooqislamicresearchcenter	34	0	78	13.83
com.objectremover.touchretouch	14	0	19	11.28
com.vwfs.phototan	3	0	5	27.23
com.sg.apphelperstore	5	0	6	6.56

Table 21. Statistics of 25 randomly selected “task-unbounded” apps from Google Play

Package name	|Act|	|Frg|	|Δ|	**Size(MB)**
org.ligi.fahrplan	6	5	46	1.83
io.gitlab.allenb1.todolist	6	3	11	1.05
naman14.timber	5	3	15	5.8
me.anon.grow	4	3	30	6.46
com.wikijourney.wikijourney	1	2	7	4.27
com.mattallen.loaned	3	1	5	1.01
com.ymber.eleven	1	2	15	10.38
com.syncedsynapse.kore2	2	1	5	2.22
net.momodalo.app.vimtouch	5	1	9	3.44
com.csipsimple	12	2	22	11.56
ch.corten.aha.worldclock	4	3	13	1.4
org.wikimedia.commons.wikimedia	3	6	44	15.0
com.llamacorp.equate	1	1	3	0.39
koeln.mop.elpeefpe	4	2	11	1.59
fr.kwiatkowski.ApkTrack	1	2	13	2.32
eu.siacs.conversations.legacy	17	2	88	11.12
com.artifex.mupdfdemo	4	2	5	25.68
org.osmdroid	15	3	28	4.62
com.haha01haha01.harail	4	1	6	0.69
com.owncloud.android	11	4	40	3.8
org.cipherdyne.fwknop2	5	2	12	2.63
edu.cmu.cylab.starslinger.demo	4	1	8	1.21
com.commonslab.commonslab	4	6	45	14.72
hans.b.skewy1_0	1	6	19	6.45
com.twobuntu.twobuntu	4	3	12	0.75

Table 22. Statistics of 25 randomly selected “fragment-container unbounded” apps from F-Droid

Package name	|Act|	|Frg|	|Δ|	**Size(MB)**
com.schoola2zlive	11	1	40	4.77
com.endless.smoothierecipes	3	9	110	7.57
com.traderumors	5	6	28	5.53
com.rakuten.room	20	19	87	20.1
com.hotels.hotelsmecca	6	1	16	16.43
br.com.prevapp03	34	1	80	5.15
com.star.mobile.video	21	1	55	7.98
fr.elol.yams	4	6	93	8.75
music.symphony.com.materialmusicv2	7	3	14	3.39
ru.sports.rfpl	6	1	9	8.33
com.discsoft.daemonsync	7	1	10	10.06
com.accuvally.android.accupass	8	1	21	11.62
com.directv.navigator	10	7	68	41.66
com.ldf.gulli.view	13	8	49	15.62
kvp.jjy.MispAndroid320	14	6	51	31.34
de.wirfahrlehrer.easytheory	3	12	144	3.6
com.mmb.mamitalk	5	4	22	8.19
ch.swissdevelopment.android	3	1	8	20.55
tw.com.iobear.medicalcalculator	1	1	3	2.49
com.gn.apkmanager	2	1	4	2.92
com.piradevrcostagen.strobo	2	1	4	2.48
com.gqueues.android.app	13	3	34	4.68
com.reverb.app	14	9	96	13.82
com.xnview.hypocam	4	8	36	26.23
de.stefanpledl.localcast	1	16	95	13.94

Table 23. Statistics of 25 randomly selected “fragment-container unbounded” apps from GooglePlay