A EFSM OF BASIC COMPONENTS

Fig. 10 displays the extended finite state machine (EFSM) for the launching of single component, which accepts the executed lifecycle methods of one component. The lifecycle method inputted into an EFSM will trigger both the changing of EFSM state and the status of a globally maintained component stack t. We use $t(i)$ to denote the i^{th} component from the top in stack t. If the current visited component is not in stack t, it is represented as $t(0)$. For example, at state S_0, the method `onCreate()` of a not stored component $t(0)$ will trigger the transition $S_0 \rightarrow S_1$ and component $t(0)$ will be pushed into t, i.e., *push* $t(0)$. After pushing, it can be obtained by getting the top element $t(1)$ from stack. At state S_1, the method `onStart()` of component $t(1)$ will trigger the transition $S_1 \rightarrow S_2$ and modify the status of component $t(1)$ to *started*, i.e., $t(1).s$ = *started*. We assume the launch mode of any component is *standard* mode.

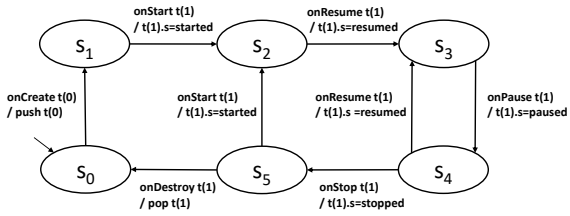

Figure 10: EFSM of Single Component

The EFSM for the launching and interactions of multiple components is given in Fig. 11, in which the red transitions are caused by multiple components interactions. For example, if component A launch B, we get a transition from $s_4 \rightarrow s_1$, then B is started and resumed ($s_1 \rightarrow s_2 \rightarrow s_3$). After that, the previous component A is stopped ($s_3 \rightarrow s_6$), or even destroyed ($s_3 \rightarrow s_6 \rightarrow s_7$). Another case is that the top component B in the stack is going to finish. So, it pauses itself and resumes the previous one ($s_4 \rightarrow s_3$), followed with the component stop and destroys operations ($s_3 \rightarrow s_6 \rightarrow s_7$). In Fig. 11, we merge the transitions with same lifecycle input for simplicity. Note that, the event subsequence (`onStop` t(1), `onPause` t(1)) could never happen in reality, as it violates the lifecycle order of single component.

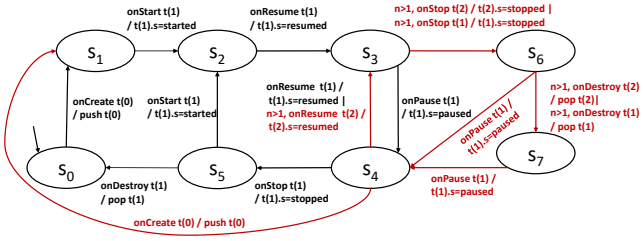

Figure 11: EFSM of Multiple Components

B ICC EXTRACTION ALGORITHM

The ICC extraction algorithm is presented in Algorithm 1. In line 1, we first preprocess a trace to remove the component switching caused by polymorphic characteristics, e.g, the invocation of `super.onCreate()`. In line 2, we build a stack model to store the visited components with a modified stack pushing mechanism. Considering the incompleteness of lifecycle methods, the callback methods are also concerned, which indirectly denotes the running status of a component. Among these methods, the stopping related methods, like `onStop()` and `onDestroy()`, are excluded as invalid operations, because the execution of them lay back to the methods in the newly launched one (refer to Fig. 11), which confuses the component launching order analysis. In lines 5-9, for an executed lifecycle or callback method, if its corresponding component is not in the stack, the component will be pushed into the stack and an ICC link will be reported. Or else, in lines 11-15, if the corresponding component already exists in the stack, that element will be moved to the top of the stack. In this case, we do not report any new ICC link. The reason is that a recently visited component is more likely reached by pressing the back button than launched by Intent. We adopt a conservative strategy to avoid FP ICCs. In order to correct the modeling error during analyzing, the stack will be cleared when the default entry component, usually is the `MainActivity` or the `SplashActivity`, is revisited and initialized.

Algorithm 1 ICC_extraction_analysis

Input: execution trace trc, up limit of stack size k
Output: extracted ICC link set Res

```
1:  removePolyCall(trc)
2:  stk = new stack()
3:  for each (cpt, mtd) in trace trc do
4:    if mtd.isValidLifecycle() or mtd.isCallback() then
5:      if cpt not in stk then
6:        stk.push(cpt, mtd)
7:        if stk.size()>1 then
8:          Res.addICC(stk.getEle(1), stk.getEle(0))
9:        end if
10:     else
11:       for i=0; i<stk.size(); i++ do
12:         if stk.getEle(i).equals(cpt) then
13:           stk.moveEletoTop(i)
14:         end if
15:       end for
16:     end if
17:     if stk.getEle(0) is the default entry component then
18:       stk.removeBottom()
19:     end if
20:   end if
21: end for
```

For example, if we have a method trace `A.onCreate()` → `B.onCreate()` → `A.onResume()` → `C.onStart()` → `A.onStop()`. We first invoke `removePolyCall()` to filter out the polymorphic call edges. If component `A` extends `B` and calls `super.onCreate()`, we remove `B.onCreate()` in the trace to avoid the FP ICC $A \rightarrow B$ and push A into stack. Or else, ICC $A \rightarrow B$ is extracted, and both A, B are pushed. If B is in the stack, when method `A.onResume()` is logged, we take A as a back operation leaded transition and do not extract ICC $B \rightarrow A$. Also, even method `A.onStop()` shows up later than `C.onStart()`, we do not report ICC from $C \rightarrow A$ because method `A.onStop()` is not a valid lifecycle in our approach.