# Static Executes-Before Analysis for Event Driven Programs

ANONYMOUS AUTHOR(S)

The *executes-before* relation between tasks is fundamental in the analysis of Event Driven Programs. We present a sound, efficient, and effective static analysis technique to compute executes-before pairs of tasks for a general class of event driven programs. The analysis is based on a small but comprehensive set of rules evaluated on a novel structure called the *task post graph* of a program. We show how to use the executes-before information to identify disjoint-blocks in event driven programs and further use them to improve the precision of data race detection for these programs. We have implemented our analysis in the Flowdroid framework in a tool called AndRacer and evaluated it on several Android apps, bringing out the scalability, recall, and improved precision of the analyses.

Additional Key Words and Phrases: static analysis, executes-before, event driven programming, race detection, asynchronous calls, Android applications

## 1 INTRODUCTION

The Event-Driven Programming (EDP) model has become a popular contemporary paradigm, widely used in the development of mobile apps, graphical user interfaces, and web applications, among others. These programs are multi-threaded programs in which each thread has associated with it a queue of program units called "tasks" that are "posted" to it by other threads, and that it executes sequentially in a FIFO manner. The posting of tasks is typically triggered by "events" like button clicks, completion of background tasks, etc. While EDP is an an efficient paradigm, control flow in these programs can be complex and non-standard, and pose a challenge to the developer to guard against common concurrency issues like data races and atomicity violations. The non-standard concurrency model also makes it challenging to carry out static analysis in a sound, precise and efficient manner.

A key notion that has proved useful in analyzing EDP programs is the "executes-before" relation on the tasks of a program. A task $a$ *executes-before* another task $b$ in an EDP program $P$ if in every execution of $P$, every instance of $a$ completes execution before any instance of $b$ begins its execution. Versions of the executes-before relation (called "happens-before" in (Hu and Neamtiu 2018) and Wu et al. (2019)) have been used to detect event-races (where the order between two events like a use and a free is not respected). Another use of the executes-before relation, which we show in this paper, is in a "disjoint block" analysis (also known as a not May-Happen-in-Parallel (not MHP) analysis) for EDP programs. Disjoint blocks are blocks of code in two tasks which are guaranteed never to overlap (or Happen-in-Parallel) in any execution of the program, much like blocks of code protected by the same lock. Disjoint block information is fundamental for data race detection (Chopra et al. 2019; Engler and Ashcraft 2003; Sterling 1993), high-level race detection for atomicity violations (Singh et al. 2019), and for identifying redundant synchronizations. A final promising use


XXXX-XXXX/2021/7-ART $15.00
https://doi.org/10.1145/1122445.1122456

of the executes-before relation appears to be in carrying out efficient data-flow analysis for EDP programs. One can imagine using executes-before information to construct a combined control-flow graph of the program and analyze it using techniques like (Chopra et al. 2019; De et al. 2011; Gotsman et al. 2007).

In this paper we propose a sound and efficient way of identifying executes-before pairs in an EDP program. We give a small set of conditions and inference rules that can be statically checked on a structure called a "task post graph" induced by the program, which are sufficient to guarantee that one task executes before another. We have implemented and evaluated the analysis on several Android apps, and observed that it has good recall of manually identified executes-before pairs in these apps.

In a further application downstream, we show how to use the executes-before information to identify pairs of disjoint-blocks in EDP programs, and apply this to statically detect data races and check for redundant synchronizations in Android apps. We show the value of the executes-before-based disjoint-block rules by observing that they contribute to 57% of the total conflicting accesses eliminated.

## 2 OVERVIEW

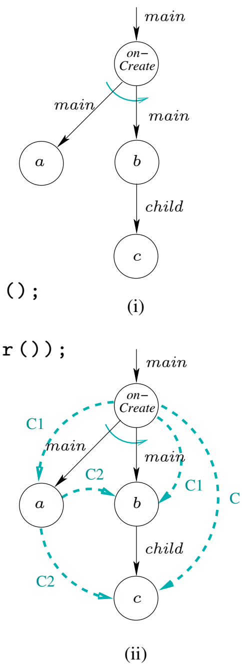
In this section we illustrate the main ideas of this paper with an example event-driven program in the form of an Android app adapted from Wu et al. (2019), shown in Fig. 1a. The figure shows an activity called `MyActivity` that has a field $p$ and four tasks `onCreate`, `a`, `b`, and `c`. When the application begins execution, the Android runtime creates the *main* thread with a FIFO queue attached to it. It then `post`'s (or enqueues) the lifecycle callback `onCreate` to the *main* thread. The *main* thread begins by dequeing the only task in its queue, `onCreate`, and executing it. Tasks in an app can post other tasks onto threads using handlers. The `onCreate` task creates a handler for *main* (line 20) using which it posts tasks `a` and `b`, in that order, onto the *main* thread's queue (lines 21–22). The *main* thread upon completion of `onCreate` proceeds with dequeing and executing task `a` which initializes the value of $p$ (line 4). The *main* thread then dequeues and executes task `b`. This task creates a *child* thread with a queue attached to it (lines 9–10) and then creates a handler to access the queue (line 11). The task then posts task `c` onto the newly created *child* thread (line 12). The *child* thread, when it gets control, dequeues and executes task `c`, which writes to variable $p$ (line 17).

We say that a task $d$ "executes-before" a task $e$ in an EDP program $P$ if whenever we have an execution of $P$ with an instance of task $d$ and $e$ respectively, the instance of $d$ must complete before the instance of $e$ begins execution. In this sense, in the given program we can see that `onCreate` executes-before both `a` and `b`. This is because firstly each task has a single instance. Secondly, `onCreate` must execute for `a` and `b` to be posted, and since they are posted to the same thread *main*, `a` and `b` must wait for `onCreate` to finish executing before they can begin execution. We can also observe that task $a$ executes before $b$ since it is posted by `onCreate` to *main* before `b` is. Finally, both `onCreate` and `a` must execute before `c` (even though they are posted to different threads) since both `onCreate` and `a` execute before `b` which posts `c`.

We now describe how we statically identify such executes-before pairs. We propose a small set of conditions ((C1), (C2) and (C3) described in Sec. 5), each of which allows us to conclude that a task executes-before another. The conditions are phrased on a structure we call a Task Post Graph (TPG), which has the set of tasks as its nodes, and an edge from task $d$ to task $e$ labelled $th$ whenever task $d$ contains a post of task $e$ to thread $th$. Fig. 1b(i) shows the TPG corresponding to the example program. The small arc arrow across the edges

```
class MyActivity extends Activity {
 long p;
 Runnable a = new Runnable() {
  public void run() {p = 0L;}
 }
 Runnable b = new Runnable() {
  public void run() {
   long y = p;
   HandlerThread child = new HandlerThread();
   child.start();
   Handler hb = new Handler(child.getLooper());
   hb.post(c);
   long z = p;
  }
 }
 Runnable c = new Runnable() {
  public void run() {p = 10;}
 }
 public void onCreate() {
  Handler handler = new Handler();
  handler.post(a);
  handler.post(b);
 }
}
```

(a) Program




(b) (i) TPG of program and (ii) with EB relation superimposed

Fig. 1. An example Android app adapted from Wu et al. (2019)

corresponding to posts of `a` and `b` from `onCreate` indicates that all posts of `a` take place before those of $b$ in `onCreate`. Fig. 1b(ii) shows the executes-before pairs inferred using these rules, using dashed edges labelled by the corresponding rule. For example, we infer that `onCreate` executes before `c` by rule (C1) (see Fig. 4(C1)(b)), which essentially says that if all paths from the initial task to `c` pass through `onCreate`, and all these paths have at least one post to the thread to which `onCreate` is posted, then `onCreate` executes-before `c`. We note that all five executes-before pairs, mentioned earlier, were inferable by our rules.

One of the uses of the executes-before information is in determining (in a sufficient way) when two blocks of code (or two tasks themselves) are "disjoint," in that they can never happen-in-parallel (or overlap in time during an execution). We give a couple of such rules in Sec. 7. The first of these rules says that if one task executes-before another they are disjoint. This lets us infer that `onCreate` is disjoint with tasks `a`, `b`, and `c`, and that `a` is disjoint with both `b` and `c`. Our second rule says that if it is the case that a parent $x$ of task $y$ executes before any other parent of $y$ then task $x$ is the *first* to post $y$, and hence the block of statements before the post of $y$ in task $x$ are disjoint with the whole of $y$. This lets us infer, in the above example, that the block of code in task `b` upto the post of `c` is disjoint from the whole of `c`.

The disjoint block information can be used to detect data races in a sound manner. To do this we first collect pairs of statements that may constitute conflicting accesses, in that they both access a common memory location, at least one of them is a write, and they may run on different threads. In the example program, the pairs of statements $(4, 17)$, $(8, 17)$, and $(13, 17)$ constitute conflicting accesses. Whenever a pair of accesses is "covered" by a pair of disjoint blocks, we can eliminate the pair as non-racy (since they can never happen-in-parallel). The access pair $(4, 17)$ is covered by the pair of disjoint tasks `a` and `c`, and hence can be eliminated. Similarly, the pair of statements $(8, 17)$ is covered by the pair of blocks comprising the first half of `b` (till the post of `c`) and the whole of `c`, and hence can be eliminated. Finally, we report $(13, 17)$ as a potentially racy pair of statements, since we were unable to eliminate it using any of our rules. We note that this pair of accesses actually constitutes a potentially harmful race.

Finally, we mention in passing another potential use of the executes-before information in carrying out efficient data-flow analysis for EDP programs. Using the executes-before pairs that we have inferred for the example program, we can construct a combined control-flow graph (CFG) of the tasks in the program, that corresponds to the expression "`onCreate`·`a`·(`b` ∥ `c`)", and carry-out a sound analysis on it. In particular, if we did an uninitialized variable analysis on the above combined CFG, we can infer that the variable $p$ is indeed initialized in the access at line 17 in task `c`.

## 3 EVENT DRIVEN PROGRAMS

An event driven program is essentially a multi-threaded program with dynamically created threads. It is organized as a set of program units called "tasks" which access a set of shared global variables. Initially there is only a "main" thread which starts off by executing a designated "main" task. Among other things, a task can create new threads and "post" tasks to other threads. Each thread conceptually maintains a FIFO queue of tasks that have been posted to it, and repeatedly dequeues and executes the task at the head of its queue. Table 1 shows the set of commands that an event driven program can use over a set of variables $V$ and locks $L$. We denote this set of commands by $Cmd_{V,L}$.

More formally an *event driven program* $P$ is a tuple $(V, L, T)$, where $V$ is a finite set of global variables, $L$ is a finite set of locks, and $T$ is a finite set of tasks. Every task $a \in T$ is represented as a *control flow graph* (CFG) $G_a = (Loc_a, ent_a, ext_a, Inst_a)$, where $Loc_a$ is the (finite) set of locations of $a$, $ent_a, ext_a \in Loc_a$ are the entry and exit locations of $a$ respectively, and $Inst_a \subseteq Loc_a \times Cmd_{V,L} \times Loc_a$ is the set of instructions of $a$. We use the notation $Inst_P = \bigcup_{a \in T} Inst_a$ to denote the set of all instructions in $P$, and $task(\iota)$ for an instruction $\iota$ in $Inst_a$ to denote the task $a$ in whose CFG it occurs. We assume a designated *main task* called $m$ in $T$, which begins the program's execution on the *main* thread. We also assume an *idle* task, which does no useful work, and executes in a thread whenever there are no other tasks to run on it. We denote the class of event driven programs by *EDP* and refer to such programs as *EDP programs*. Fig. 2 shows the textual version of an example EDP program with 3 tasks: $m$, $count$, and $prod$.

Before we define the semantics of an EDP program, some notations will be useful. We use $\mathbb{Z}$ to denote the set of integers. We denote the set of finite sequences (or *words*) over a finite set of symbols $S$ by $S^*$, and represent the empty sequence by $\epsilon$. For a function $f : A \to B$, $a \in A$ and $b \in B$, we use $f[a \mapsto b]$ to denote the function $g : A \to B$ given by $g(x) = f(x)$ for $x \neq a$ and $g(x) = b$ otherwise. If $C \subseteq A$, we use $f \upharpoonright C$ to denote the restriction of $f$ to the domain $C$. For a logical condition $b$ over a set of variables $V$ we denote by $[\![b]\!]$ the set of valuations that satisfy $b$. For an arithmetic expression $e$ over variables $V$, and a valuation $\phi$ for $V$, we denote by $[\![e]\!]_\phi$ the value obtained by evaluating $e$ in $\phi$.

Some general notions for rooted labelled directed graphs will be useful going forward. We represent such a graph by a tuple $G = (V, r, \Sigma, E)$, where $V$ is the set of nodes of the graph,

Table 1. EDP Program Commands $Cmd_{V,L}$

| Statement | Description |
|---|---|
| $t :=$ `create`() | Create a new thread and store the thread id in $t$. |
| `stopth`() | Stop executing the current thread. |
| `join`($t$) | Current thread waits until thread $t$ finishes executing. |
| `post`($t$,$a$) | Enqueue task $a$ on to thread $t$'s queue. |
| `skip` | Do nothing. |
| $x := e$ | Assign the value of expression $e$ to variable $x$. |
| `assume`($b$) | Enabled only if expression $b$ evaluates to *true*; does nothing. |
| `lock`($l$) | Current thread takes lock $l$ if available; otherwise blocks till $l$ is available. |
| `unlock`($l$) | Current thread releases lock $l$. |

$r \in V$ is a designated root node, $\Sigma$ is the set of edge labels, and $E \subseteq V \times \Sigma \times V$ is the set of labelled directed edges of the graph. Let $G = (V, r, \Sigma, E)$ be a labelled directed graph. A *path* from node $u$ to $v$ is a finite (possibly empty) sequence of connected edges in the graph, starting at $u$ and ending at $v$. The *length* of a path is the number of edges in the path. Given a label $\sigma \in \Sigma$, we define the $\sigma$-*length* of a path $\pi$ in $G$ to be the number of $\sigma$-labelled edges in $\pi$. We say a node $m$ *dominates* another node $n$ in $G$, denoted $dom(m, n)$, if every path from the root node $r$ to $n$ passes through $m$.

Let $P = (V, L, T)$ be an EDP program. We define the semantics of $P$ as a labelled transition system $\mathcal{S}_P = (S, s_0, \delta)$, where $S$ is the set of states, $s_0 \in S$ is the initial state, and $\delta$ is the transition relation, as described below.

A state $s \in S$ is a tuple $\langle \mathcal{T}, \mathcal{Q}, M_T, M_\mathcal{Q}, M_C, M_L, \phi \rangle$, where

- $\mathcal{T}$ is a set of active threads (that are created but not terminated),
- $\mathcal{Q}$ is a set of queues, one for each thread in $\mathcal{T}$,
- $M_T : \mathcal{T} \to T \times Loc$ associates with each active thread a task and a location in the task, representing its current location. Thus if $M_T(th) = (t, l)$, then we require that $l \in Loc_t$.
- $M_\mathcal{Q} : \mathcal{T} \to \mathcal{Q}$ associates with each thread a queue,
- $M_C : \mathcal{Q} \to T^*$ associates with each queue a sequence of tasks representing the current contents of the queue,
- $M_L : L \rightharpoonup \mathcal{T}$ is a partial map which associates with each lock the thread (if any) that has acquired the lock, and
- $\phi : V \to \mathbb{Z}$ is a valuation for variables representing their current value.

The initial state is given by

$$s_{in} = (\{main\},\ \{mainQueue\},\ \lambda thd.(m, ent_m),\ \lambda thd.mainQueue,\ \lambda q.\epsilon,\ undef,\ \lambda x.0).$$

The transition relation $\delta$ describes the possible transitions between states, and captures the semantics of the program. Let $s = (\mathcal{T}, \mathcal{Q}, M_T, M_\mathcal{Q}, M_C, M_L, \phi)$ and $s' = (\mathcal{T}', \mathcal{Q}', M'_T, M'_\mathcal{Q}, M'_C, M'_L, \phi')$ be two states, and $\iota = (l, c, l')$ be an instruction in a task $a$, with $l' \neq ext_a$. Then we have $(s, \iota, s') \in \delta$ iff there exists a thread $t$ in $\mathcal{T}$ such that $M_T(t) = (a, l)$, and either:

- $c$ is the command `skip`, $\mathcal{T}' = \mathcal{T}$, $\mathcal{Q}' = \mathcal{Q}$, $M'_T = M_T[t \mapsto (a, l')]$, $M'_\mathcal{Q} = M_\mathcal{Q}$, $M'_C = M_C$, $M'_L = M_L$, and $\phi' = \phi$; or
- $c$ is the command `assume`($b$), $\phi \in [\![b]\!]$, $\mathcal{T}' = \mathcal{T}$, $\mathcal{Q}' = \mathcal{Q}$, $M'_T = M_T[t \mapsto (a, l')]$, $M'_\mathcal{Q} = M_\mathcal{Q}$, $M'_C = M_C$, $M'_L = M_L$, and $\phi' = \phi$; or

- $c$ is the command $x := e$, $\mathcal{T}' = \mathcal{T}$, $\mathcal{Q}' = \mathcal{Q}$, $M'_T = M_T[t \mapsto (a, l')]$, $M'_\mathcal{Q} = M_\mathcal{Q}$, $M'_C = M_C$, $M'_L = M_L$, and $\phi' = \phi[x \mapsto [\![e]\!]_\phi]$; or
- $c$ is the command `stopth`, $\mathcal{T}' = \mathcal{T} - \{t\}$, $\mathcal{Q}' = \mathcal{Q} - \{M_\mathcal{Q}(t)\}$, $M'_T = M_T \restriction \mathcal{T}'$, $M'_\mathcal{Q} = M_\mathcal{Q} \restriction \mathcal{T}'$, $M'_C = M_C \restriction \mathcal{Q}'$, $M'_L = M_L$, and $\phi' = \phi$; or
- $c$ is the command $th :=$ `create`(), $\mathcal{T}' = \mathcal{T} \cup \{tid\}$ for some $tid \notin \mathcal{T}$, $\mathcal{Q}' = \mathcal{Q} \cup \{qid\}$ for some $qid \notin \mathcal{Q}$, $M'_T = M_T[t \mapsto (a, l')][tid \mapsto (idle, ent_{idle})]$, $M'_\mathcal{Q} = M_\mathcal{Q} \cup \{tid \mapsto qid\}$, $M'_C = M_C \cup \{qid \mapsto \epsilon\}$, $M'_L = M_L$, and $\phi' = \phi[th \mapsto tid]$; or
- $c$ is the command `join`($th$), $\phi(th) \notin \mathcal{T}$, $\mathcal{T}' = \mathcal{T}$, $\mathcal{Q}' = \mathcal{Q}$, $M'_T = M_T[t \mapsto (a, l')]$, $M'_\mathcal{Q} = M_\mathcal{Q}$, $M'_C = M_C$, $M'_L = M_L$, and $\phi' = \phi$; or
- $c$ is the command `post`($th, b$), $\phi(th) \in \mathcal{T}$, $M_C(q) \neq \epsilon$, $\mathcal{T}' = \mathcal{T}$, $\mathcal{Q}' = \mathcal{Q}$, $M'_T = M_T[t \mapsto (a, l')]$, $M'_\mathcal{Q} = M_\mathcal{Q}$, $M'_C = M_C[q \mapsto (M_C(q) \cdot b)]$, $M'_L = M_L$, and $\phi' = \phi$, where $q = M_\mathcal{Q}(\phi(th))$; or
- $c$ is the command `post`($th, b$), $\phi(th) \in \mathcal{T}$, $M_C(q) = \epsilon$, $M_T(\phi(th)) = (idle, -)$, $\mathcal{T}' = \mathcal{T}$, $\mathcal{Q}' = \mathcal{Q}$, $M'_T = M_T[t \mapsto (a, l')][\phi(th) \mapsto (b, ent_b)]$, $M'_\mathcal{Q} = M_\mathcal{Q}$, $M'_C = M_C$, $M'_L = M_L$, and $\phi' = \phi$, where $q = M_\mathcal{Q}(\phi(th))$; or
- $c$ is the command `lock`($k$), $M_L(k)$ is undefined, $\mathcal{T}' = \mathcal{T}$, $\mathcal{Q}' = \mathcal{Q}$, $M'_T = M_T[t \mapsto (a, l')]$, $M'_\mathcal{Q} = M_\mathcal{Q}$, $M'_C = M_C$, $M'_L = M_L[k \mapsto t]$, and $\phi' = \phi$; or
- $c$ is the command `unlock`($k$), $M_L(k) = t$, $\mathcal{T}' = \mathcal{T}$, $\mathcal{Q}' = \mathcal{Q}$, $M'_T = M_T[t \mapsto (a, l')]$, $M'_\mathcal{Q} = M_\mathcal{Q}$, $M'_C = M_C$, $M'_L = M_L - \{(k, t)\}$, and $\phi' = \phi$.

For the case when $l' = ext_a$, the rules are similar, except that the thread $t$ now switches to $(b, ent_b)$ when $t$'s queue is non-empty and $b$ is the task at the head of $t$'s queue; when $t$'s queue is empty, $t$ will now point to $(idle, ent_{idle})$.

An *execution* of an event driven program $P$ is a finite sequence of transitions $\rho = \tau_1, \ldots, \tau_n$ ($n \geq 1$) of $\mathcal{S}_P$, such that there exists a sequence of states $s_0, \ldots, s_n$ of $\mathcal{S}_P$, with each $\tau_i$ of the form $(s_{i-1}, \iota_i, s_i)$ for some $\iota_i$, and $s_0 = s_{in}$. The sequence of instructions executed in $\rho$ is $\iota_1, \ldots, \iota_n$.

It is convenient to visualize an execution of an EDP program as a sequence of instructions (or statements), with time going downwards and a column for each thread, as shown in Fig. 2. Note that there may be multiple *instances* of a task that execute in the same or different threads in an execution. In the example execution of Fig. 2 the task *count* has three instances, two in the *main* thread and one in the *child* thread. However, each instance (except possibly the last one on a thread) runs to *completion* in that once the instance is executing on a thread, it is not switched out from the thread until it completes by reaching its exit location. If we project an execution to a single thread *th* it will look like a sequence of initial and complete execution paths (except possibly for the last one which may only be initial) through the CFGs of the different tasks.

We close this section with some notions related to task CFGs. Let $P = (V, L, T)$ be an EDP program, and let $a$ be a task in $T$. Let $\iota' = (l, c, l')$ and $\iota = (m, c, m')$ be instructions in $Inst_a$. We say instruction $\iota' = (l, c, l')$ *may follow* instruction $\iota$ if there is a path from $m'$ to $l$ in $G_a$. We say $\iota$ *dominates* $\iota'$ if every path from $ent_a$ to $l'$ passes through $m$.

## 4 TASK POST GRAPH

In this section we introduce the *Task Post Graph* (TPG) structure for an event-driven program. This structure will help us in identifying executes-before pairs in an EDP program in a structural manner.

The TPG of an EDP program $P$ contains information about task $a$ possibly posting task $b$ to a thread $th$, represented by an edge in the graph from $a$ to $b$ labelled $th$. Note however

```
task m:                 task count:      task prod:

1. n := 0;              8. n := n + x;   10. x := x + 1;
2. x := 0;              9. x := 0;       11. post(child, count);
3. post(main, count);
4. child := create();
5. post(child, prod);
6. x := x + 1;
7. post(main, count);
```

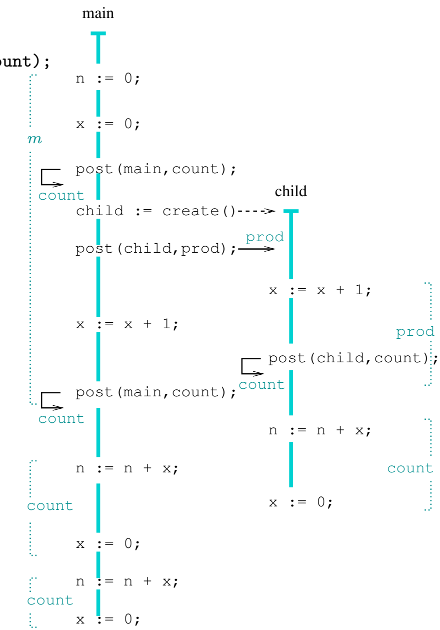


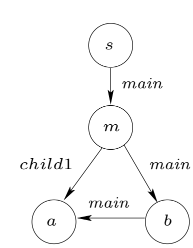

Fig. 2. An example EDP program $P_1$ and one of its executions

that tasks may be posted to "concrete" threads created *dynamically* during the execution of $P$. To use a static label for the post edges, we make use of the notion of *abstract* threads. We associate all the threads created at a particular `create` statement in the program with an "abstract" thread corresponding to that statement. For convenience we assume that in an EDP program a thread variable is assigned at only one statement, and we use the thread variable as the name of the abstract thread associated with that `create` statement. We note that a `create` statement in $P$ may be executed *multiple* times during an execution of $P$, as it may be in a loop in a task, or it may be in a task that is posted multiple times during the execution of $P$. We say an abstract thread is *unique* if it corresponds to exactly one concrete thread. For convenience we call such an abstract thread a *unique thread.*

```
task m:                     task a:

1. child1 := create();      10. child3 := create();
2. post(child1, a);
3. post(main, b);           b:
4. while (*)                20: post(main, a);
5.   child2 := create();
```

(a) Program $P_2$




(b) Task Post Graph of $P_2$

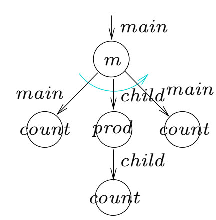
Fig. 3. An example program illustrating abstract threads and its TPG

To illustrate these notions, consider the example program $P_2$ of Fig. 3a. There are four abstract threads: *child1*, *child2*, *child3*, and the implicitly created thread *main*. A concrete thread, which is assigned to the variable *child1*, is created at line 1 of task $m$ and the corresponding abstract thread is *child1*. Both the abstract threads *main* and *child1* are unique. The abstract threads *child2* and *child3* due to lines 5 and 10, respectively, are *not* unique. For the case of abstract thread *child2*, this is due to the creation of concrete threads in a loop. For the case of abstract thread *child3*, this is due to multiple posts of task $a$, that creates a thread at line 10, from different locations (lines 2 and 20).

Let $P = (V, L, T)$ be an EDP program. The *task post graph (TPG)* induced by $P$, denoted $TPG_P$, is a labelled directed graph $(N, E)$ where $N = T \cup \{s\}$ is the set of vertices of the graph corresponding to the tasks of $P$ and a "dummy" initial vertex $s$, and $E$ is the set of labelled edges of the form $(a, th, b)$ such that task $a$ contains a post of task $b$ to the abstract thread $th$ in $P$. We also add the edge $(s, main, m)$ in $E$ to denote the implicit posting of the main task $m$ to the *main* thread. The TPG for the program $P_2$ in Fig. 3a is shown in Fig. 3b. To avoid clutter, hereafter, we leave out the dummy node $s$ from the diagrammatic representation of the TPG.

Next we define a few notions related to the task post graph that will be useful in the sequel.

*Instance Post Tree.* The instance post tree corresponding to an execution of an EDP program depicts the different task instances that were created during the execution and the order in which one instance posted other task instances to (abstract) threads. More formally, let $P = (V, L, T)$ be an EDP program, and let $\rho$ be an execution of $P$. The *instance post tree* corresponding to $\rho$, denoted $IPT_\rho$, is a rooted directed ordered tree with nodes corresponding to task instances in $\rho$, the first instance of $m$ as the root, and labelled edges $(i, th, j)$ whenever task instance $i$ posts task instance $j$ to the abstract thread $th$. Moreover for each instance $i$ the children of $i$ are *ordered* according to the order in which they were posted in $i$. The figure alongside shows the instance post tree corresponding to the execution shown in Fig. 2, with the children of a node being ordered from left to right (the blue arc also indicates this). We note that every path in the instance post tree of an execution $\rho$ of $P$ is also a path in $TPG_P$ (essentially the tree $IPT_\rho$ embeds homomorphically into $TPG_P$).




We say that an edge from task $a$ to task $b$ labelled $th$ in $TPG_P$ is a *unique post edge* if there is exactly one `post`$(th, b)$ statement in $a$, and moreover that statement is not in a loop. It is easy to see that if $(a, th, b)$ is a unique post edge, then any instance of $a$ can post at most one instance of $b$ to thread $th$.

We say that a task $a$ in $P$ is *unique* if every execution of $P$ contains at most one instance of $a$. A sufficient condition on $TPG_P$ that ensures that task $a$ is unique is that there should be a unique path from $m$ to $a$, and all edges along this path should be unique post edges (in the sequel we will refer to this condition as "a unique path of unique posts"). To see that the condition is indeed sufficient, suppose we had two instances of $a$ in an execution $\rho$ of $P$, and consider the instance post tree $IPT_\rho$ of $\rho$. Consider the two paths $\pi$ and $\pi'$ from $m$ to the two instances of $a$ in this tree, and let $x$ be the lowest common ancestor of the two instances of $a$ along these paths. Let $x$ be an instance of task $b$. Let $y$ and $y'$ be the two children of $x$ along the paths $\pi$ and $\pi'$ respectively. If $y$ and $y'$ are instances of different tasks, then we do not have a unique path from $m$ to $a$ in $TPG_P$. If $y$ and $y'$ are instances of the same task say $c$, then the $(b, th, c)$ edge in $TPG_P$ cannot be a unique post edge.

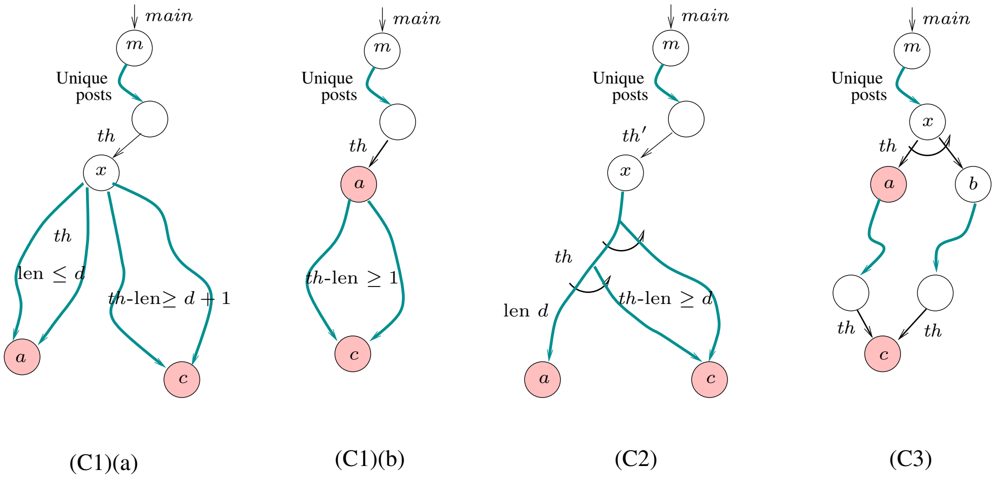



Fig. 4. Illustrating sufficient conditions (C1)–(C3) on the TPG of a program, for $a$ to execute before $c$.

*Order between paths.* Let $\pi$ and $\pi'$ be two paths in the TPG of a program $P$. We say $\pi$ is *ordered-before* $\pi'$ if $\pi = \pi_1 \cdot (x, th, y) \cdot \pi_2$ and $\pi' = \pi_1 \cdot (x, th', z) \cdot \pi_2'$ for some paths $\pi_1$, $\pi_2$, and $\pi_2'$, threads $th$ and $th'$, and tasks $x$, $y$ and $z$, such that $y \neq z$, $\pi_2$ and $\pi_2'$ have no node in common, and each post of task $y$ dominates all posts of task $z$ in the CFG of task $x$.

## 5 EXECUTES-BEFORE

In this section, we describe sufficient conditions for when a task is guaranteed to "execute before" another task in an EDP program.

Let $P$ be an EDP program, and let $a$ and $c$ be tasks in $P$. We say task $a$ *executes before* task $c$ in $P$, if in every execution $\rho$ of $P$, *every* instance of $a$ completes execution before *any* instance of $c$ begins execution in $\rho$. More precisely, suppose $\rho$ contains the entry instruction of an instance of $c$ at position $j$ and the entry instruction of an instance of $a$ at position $i$; then $i < j$ and there exists a position $k$ with $i < k < j$, such that the instance of $a$ executes its exit instruction at position $k$.

We describe several sufficient conditions on an EDP program and its TPG, which will ensure that a certain task executes before another. Let $P = (V, L, T)$ be an EDP program, and $a$ and $c$ two distinct tasks in $T$. Each condition on $TPG_P$ below aims to ensure that $a$ executes before $c$. Figs. 4 and 5 illustrate these conditions. In the figures, an arc arrow across path $\pi$ and $\pi'$ indicates that $\pi$ is ordered-before $\pi'$.

(C1) This condition is illustrated in Fig. 4(C1)(a). There is a task $x$ which is posted to a unique thread $th$, and a number $d \geq 0$ such that:
  (1) There is a unique path of unique posts from $m$ to $x$;
  (2) All paths from $m$ to $a$ and $m$ to $c$ pass through $x$;
  (3) Each path from $x$ to $a$ is labelled $th$ and has length at most $d$; and
  (4) Every path from $x$ to $c$ has $th$-length at least $d + 1$.
  Fig. 4(C1)(b) shows the special case of this condition when $d = 0$ and $a = x$.

(C2) This condition is illustrated in Fig. 4(C2). There is a task $x$, a unique thread $th$, and a number $d \geq 1$, such that:
  (1) There is a unique path of unique posts from $m$ to $x$;

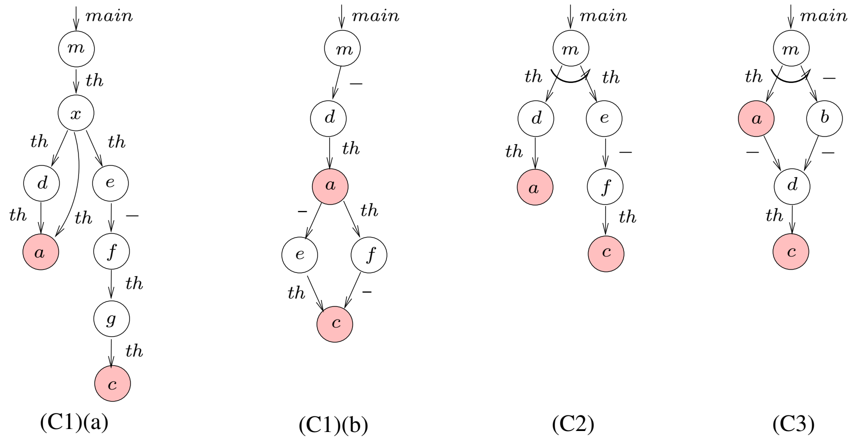



Fig. 5. Example TPGs of programs satisfying conditions (C1)–(C3). In each case task $a$ executes before $c$.

(2) All paths from $m$ to $a$ and $m$ to $c$ pass through $x$;
(3) There is a unique path $\pi$ of unique posts of length $d$ from $x$ to $a$, with all edges labelled $th$; and
(4) For every path $\pi'$ from $x$ to $c$:
  – The path $\pi'$ is ordered after the path $\pi$ from $x$ to $a$; and
  – The $th$-length of $\pi'$ is at least $d$.

(C3) This condition is illustrated in Fig. 4(C3). There is a task $x$ and a unique thread $th$ such that:
(1) There is a unique path of unique posts from $m$ to $x$;
(2) $x$ posts task $a$ onto $th$ via a unique post, and is the only task to post $a$.
(3) For every child $b$ of $x$ other than $a$, the path from $m$ to $a$ should be ordered-before a path from $m$ to $b$.
(4) All paths from $m$ to $c$ pass through $x$.
(5) Task $c$ is always posted to the thread $th$;

Fig. 5 shows the TPGs of some EDP programs that satisfy the conditions (C1)–(C3) respectively. The edge label "-" indicates that the thread does not matter. In each case the task $a$ can be seen to execute before task $c$.

Next we define some ways of inferring executes-before pairs from an initial set of such pairs in $P$.

(I1) If a task $a$ executes before every parent $d$ of a task $c$ in $TPG_P$, then $a$ must execute before $c$. (See Fig. 6(I1)).

(I2) If tasks $a$ and $c$ are such that
(1) There is a unique path of unique posts from $m$ to $a$ in $TPG_P$,
(2) $a$ is posted to a unique thread $th$,
(3) $a$ posts $c$ to $th$, and
(4) $a$ executes before every parent of $c$ that is different from $a$;
then $a$ must execute before $c$ (See Fig. 6(I2)).

(I3) If tasks $a$, $d$, and $c$ are such that $a$ executes before $d$ and $d$ executes before $c$, then $a$ must execute before $c$. (See Fig. 6(I3)).

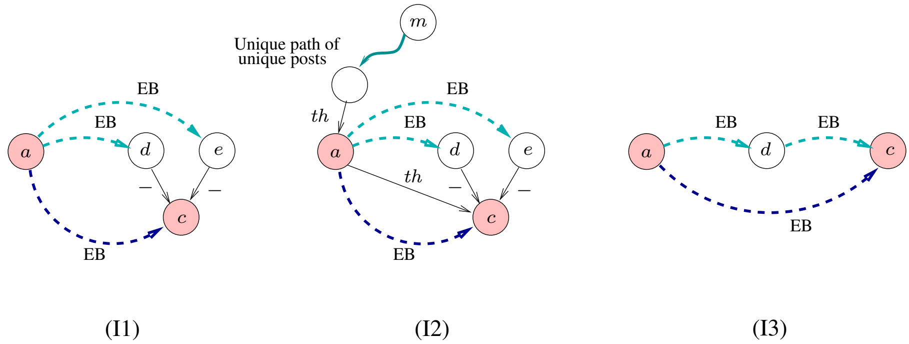



Fig. 6. Illustrating the executes-before inference rules (I1), (I2) and (I3). The dark arrows are the edges added by the rules.

We can now give a simple saturation algorithm (Algo. 1), to compute a sound set of executes-before pairs.

---

**Algorithm 1:** Compute EB pairs

---

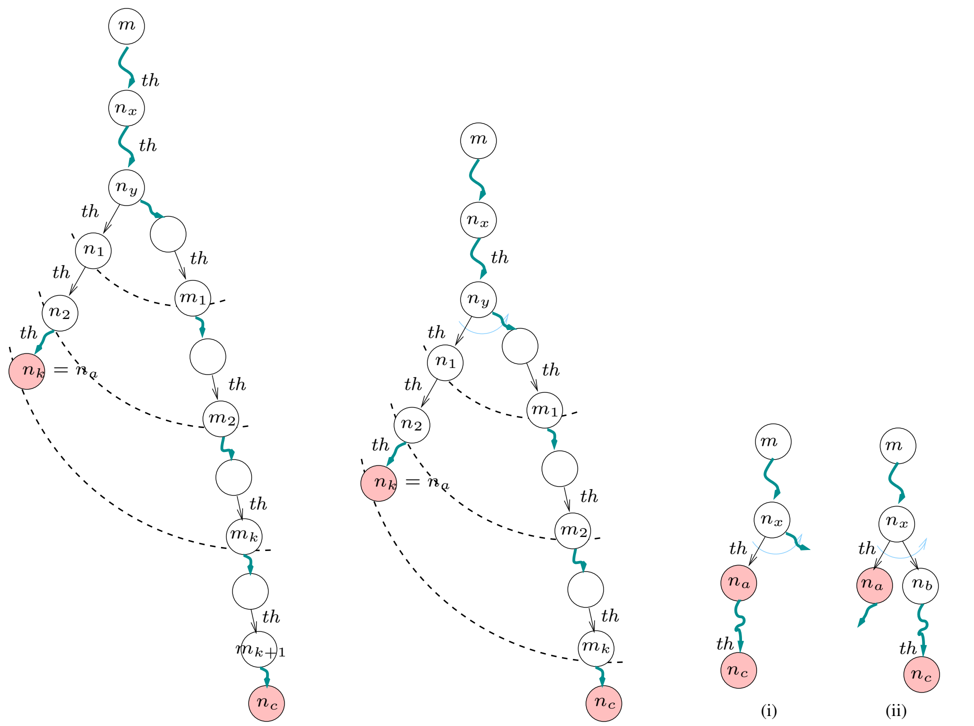
**Data:** EDP Program $P$
**Result:** Set $EB$ of executes-before pairs
$EB := \emptyset$;
Add pairs $(a, c)$ to $EB$ based on conditions (C1)–(C3);
**while** $\exists$ *a new pair* $(a, c)$ *that can be inferred by rules (I1)–(I3)* **do**
  $EB := EB \cup \{(a, c)\}$;
**end**
**return** $EB$;

---

THEOREM 5.1. *The set EB returned by Algo. 1 for an EDP program $P$ is sound in that if $(a, c) \in EB$ then $a$ executes before $c$ in $P$.*

*Proof.* It is sufficient to argue that (a) the base rules (C1)-(C3) are sound, and that (b) the inference rules (I1)–(I3) are sound as well.

To see the soundness of rule (C1), let $a$ and $c$ be tasks in program $P$ satisfying the conditions of the rule, and consider an execution $\rho$ of $P$ containing two instances of $a$ and $c$. Consider the instance post tree $IPT_\rho$ of $\rho$, and let $n_a$ and $n_c$ be the nodes corresponding to the above instances of $a$ and $c$ respectively. Let $\pi$ and $\pi'$ be the two initial paths in the tree to $n_a$ and $n_c$ respectively. We note that $\pi$ and $\pi'$ must correspond to initial paths in $TPG_P$. By the conditions of (C1), the two paths in $IPT_\rho$ must appear as shown in Fig. 7a (except possibly for the left-to-right ordering). Here $n_x$ is posted to $th$, $n_y$ is the lowest common ancestor of $n_a$ and $n_c$ in the tree, and all tasks from $n_x$ to $n_y$ are posted onto $th$. Let $n_y = n_0, n_1, \ldots, n_k = n_a$ $(k \geq 0)$ be the task instances in the path from $n_y$ to $n_a$ (all these task instances being posted to $th$). Then there must exist a subsequence of task instances $m_1, \ldots, m_{k+1}$ in the path from $n_y$ to $n_c$ (excluding $n_y$), such that each $m_i$ is posted to $th$. The dashed contours in the figure indicate the same $th$-level from $n_y$ (or $n_x$). It is clear that $n_0 = n_y$ must be posted to $th$ before $m_1$ is. We can now argue that $n_1$ must be posted to $th$ before $m_2$ is. Either $n_1$ is posted to $th$ before $m_1$ is, or $m_1$ is posted before




(a) Instance post tree for (C1) (b) Instance post tree for (C2) (c) Instance post tree for (C3)

Fig. 7. Illustrating the soundness argument for (C1), (C2), and (C3)

$n_1$ is. In the former case it is clear that $n_1$ would be posted before $m_2$. In the latter case, $m_1$ must wait for $n_y$ to complete its execution on $th$ before it can post $m_2$, by when $n_y$ would have posted $n_1$ to $th$. Thus in both cases, $n_1$ is posted to $th$ before $m_2$ is. It now follows that $n_2$ must be posted to $th$ before $m_3$ is; and so on, till $n_a = n_k$ is posted to $th$ before $m_{k+1}$ is. Since they are posted to the same unique thread $th$, $n_a$ must finish execution before $m_{k+1}$ can begin execution. Since $n_c$ can be posted only after $m_{k+1}$ begins execution, it follows that the instance $n_a$ must complete its execution before the instance of $n_c$ begins execution. This proves that $a$ must execute before $c$ in $P$.

For the soundness of (C2), consider tasks $a$ and $c$ satisfying the conditions of (C2), and consider an execution $\rho$ containing an instance of $a$ and $c$. Once again the instance post tree of $\rho$ must look like the one shown in Fig. 7b. By the ordering condition, the post of the task corresponding to $n_1$ to $th$ in the instance $n_y$ of $y$ must have taken place before the post of the task (say $z$) that leads to the post of $m_1$. Thus $n_1$ is in the queue of $th$ before the instance of $z$ is posted, and therefore before $m_1$ is eventually posted to $th$. Continuing this argument, we have that $n_k = n_a$ is posted to $th$ before $m_k$ is; and hence $n_a$ completes its execution on $th$ before $m_k$ begins, and hence also before $n_c$ begins. This proves that $a$ executes before $c$.

For the soundness of (C3), consider an execution $\rho$ of the program $P$ containing instances of $a$ and $c$. Using the $IPT_\rho$, here we show an instance of $a$ completes execution before an instance of $c$ can even start.

Let $n_a$ and $n_c$ be the nodes in the $IPT_\rho$, corresponding to the instances of $a$ and $c$ respectively. In $IPT_\rho$, due to constraints (1) and (2) of rule (C3), there is exaxtly one instance of the tasks in the path from $m$ to $a$ in $TPG_P$. Thus $n_a$ is the only instance of task $a$ while task $c$ can have multiple instances, $n_c$ being one of them. Further, from constraints (2) and (4) in (C3), node $n_c$ can be (i) a descendant of $n_a$ in $IPT_\rho$, or (ii) a descendant of $n_b$ in $IPT_\rho$, where $b$ is a sibling of $a$ in $TPG_P$ (as in Fig. 7c).

Case (i): It is easy to see that the instance $n_a$ is posted even before $n_c$ is posted. Since they are posted to the same unique thread $th$ (due to constraints (1) and (5) of (C3)), instance $n_a$ completes execution even before $n_c$ can start.

Case (ii): Instance $n_c$ is posted only after $n_b$ is posted. Since the path from task $m$ to task $a$ is ordered before any path from $m$ to $b$ (due to constraint (3)), instance $n_x$ posts $n_a$ even before it posts $n_b$. Since $n_a$ and $n_c$ are posted to the same unique thread $th$, $n_a$ appears in $th$'s queue even before $n_c$. Thus instance $n_a$ completes execution before $n_c$ can even start.

In either case, instance $n_a$ of task $a$ completes execution before an instance $n_c$ of task $c$. Thus $a$ executes before $c$.

Coming now to the soundness of the inference rules (I1)–(I3). Consider rule (I1), and suppose tasks $a$ and $c$ satisfy the conditions of the rule in $P$. Consider an execution $\rho$ with an instance of $a$ and $c$. Now the instance of $c$ must have been posted by one of the parents $d$ of $c$. But $a$ executes before $d$, so the instance of $a$ must have completed before the instance of $d$ began, and hence before the instance of $c$ began. For the case of (I2), suppose tasks $a$ and $c$ satisfy the conditions of rule (I2), and consider an execution $\rho$ with an instance of $n_a$ of $a$ and $n_c$ of $c$. If $n_c$ was posted by a task different from $a$, then similar to the previous argument $n_a$ would execute before $n_c$. If $n_c$ was posted by an instance of task $a$, then since $a$ has at most one instance by the conditions of (I2), $n_c$ must have been posted by $n_a$ to $th$. Since $th$ is a unique thread, $n_c$ can only execute once $n_a$ has finished. This completes the soundness argument for (I2). The soundness of rule (I3) is immediate. □

## 6 DATA RACES AND MAY HAPPEN IN PARALLEL

In this section we define data races and introduce "may happen in parallel" notions for EDP programs.

Let us fix an EDP program $P = (V, L, T)$. Consider two tasks $a$ and $b$ in $T$ ($a$ and $b$ could be the same task), and two non-empty paths $\pi$ and $\pi'$ in $G_a$ and $G_b$ respectively. We say $\pi$ and $\pi'$ *may happen in parallel* in $P$ if there is an execution $\rho$ of $P$, and two instances of $a$ and $b$ in $\rho$, in which the paths $\pi$ and $\pi'$ interleave (that is, either $\pi'$ begins after $\pi$ has begun but not yet ended; or vice-versa).

We now define when two statements $s_1$ and $s_2$ (corresponding, say, to instructions $\iota_1 = (l_1, c_1, l_1')$ and $\iota_2 = (l_2, c_2, l_2')$) in tasks $a$ and $b$ in $P$ respectively, "may happen in parallel." Consider the program $P'$ obtained from $P$ by enclosing the statements $s_1$ and $s_2$ in `skip` statements. More formally, we obtain $P'$ by replacing the instruction $\iota_1$ by the sequence of instructions $(l_1, \texttt{skip}, m_1)$, $(m_1, c_1, m_1')$, and $(m_1', \texttt{skip}, l_1')$, where $m_1$ and $m_1'$ are new locations in $Loc_a$; and similarly for $\iota_2$. Let $\pi_1$ be the path $l_1 \xrightarrow{\texttt{skip}} m_1 \xrightarrow{c_1} m_1' \xrightarrow{\texttt{skip}} l_1'$ in $G_{a'}$, and similarly $\pi_2$ in $G_{b'}$. We now say $s_1$ and $s_2$ *may happen in parallel* in $P$, if the paths $\pi_1$ and $\pi_2$ may happen in parallel in the program $P'$. In the example program of Fig. 2, statements in lines 6 and 10 may happen in parallel, whereas statements in lines 2 and 10 *cannot* happen in parallel.

Two statements are called *conflicting accesses* if they are read/write accesses to the same variable, at least one of them is a write, and the two statements may run on different threads.

We say two statements $s_1$ and $s_2$ in $P$ are involved in a *data race* (or are simply *racy*) if they are conflicting accesses that may happen in parallel. Thus, the statements 6 and 10 in the example program of Fig. 2 are racy, but statements 2 and 10 are not. Similarly, statement 8 races with itself, while statement 10 does not.

Finally, we define what it means for a "block" of code to happen in parallel with another. A *block* of code in $P$ is specified by a pair $(l, X)$, where for some task $a$ in $P$, $l$ is a location in $Loc_a$ and $X \subseteq Loc_a$ is a subset of locations reachable from $l$, in task $a$. An *initial path* in a block $B = (l, X)$ of a task $a$ in $P$, is a non-empty path in $G_a$ that begins at $l$ and stays within the set of locations $X$, except possibly for the last location in the path. We say a statement $s = (m, c, m')$ in $P$ *belongs to* block $B = (l, X)$ if $m$ belongs to the set $X$. We say two blocks $B_1$ and $B_2$ of $P$ *may happen in parallel* if there are two initial paths $\pi_1$ in $B_1$ and $\pi_2$ in $B_2$, which may happen in parallel with each other. Otherwise, we say $B_1$ and $B_2$ are *disjoint*. In the example program of Fig. 2, $B_1 = (1, \{1, 2\})$ and $B_2 = (10, \{10, 11\})$ are blocks in tasks `m` and `prod`, respectively. The two blocks can be seen to be disjoint.

We observe that if $s_1$ and $s_2$ are statements in two blocks $B_1$ and $B_2$ respectively in $P$, and $B_1$ and $B_2$ are disjoint with each other, then it follows that $s_1$ and $s_2$ *cannot* happen in parallel.

## 7 DISJOINT BLOCK RULES

In this section we present four rules to identify pairs of disjoint blocks in an EDP program. The first two are novel and are based on the executes-before order in the program, while the last two based on fork/join and locks are more standard.

Let us fix an EDP program $P = (V, L, T)$ for the rest of this section. Let $a$ and $b$ be two tasks in $T$. The rules below tell us when $a$ (or a part of it) is disjoint from $b$.

(Rule 1) ("First-To-Post") *Let $a$ and $b$ be tasks in $P$ such that $a$ is a unique task, $a$ posts $b$, and $a$ executes before every other parent $d$ of $b$. Let $X = Loc_a \setminus \{n \in Loc_a \mid n \text{ may follow a post of } b \text{ in } a\}$. Then the blocks $(ent_a, X)$ and $b$ are disjoint.*

(Rule 2) ("Executes-Before") *Let $a$ and $b$ be tasks in $P$ such that $a$ executes before $b$. Then the tasks $a$ and $b$ are disjoint.*

(Rule 3) ("Join") Let $a$ be a task with a `join`($th'$) statement in it. Then the block $B$, shown in Fig. 9a, is disjoint with the task $b$ if

(1) $th'$ corresponds to a unique abstract thread.
(2) For every $p$, parent of $b$, $(p, b) \in E$ is labeled with $th'$.

Observe that task $c$ that is posted after the `join` statement is disjoint with task $b$, provided only task $a$ posts $c$. Any task $d$ that is, in turn, posted by $c$ is disjoint with $b$ provided $a$ dominates $d$ in the TPG.

(Rule 4) ("Lock") Two blocks $B_1$ and $B_2$, as in Fig. 9b, enclosed in `lock`($l$)-`unlock`($l$) statements are disjoint.

In Fig. 8 we illustrate the application of the first couple of rules to some example program TPGs. The dashed arrows represent the EB relation computed by our algorithm, and the figures below show the disjoint pairs of tasks computed by the rules. Part (a) of the figure shows an application of Rule 1 to show that the "pre-post($b$)" part of task $a$ is disjoint from $b$.

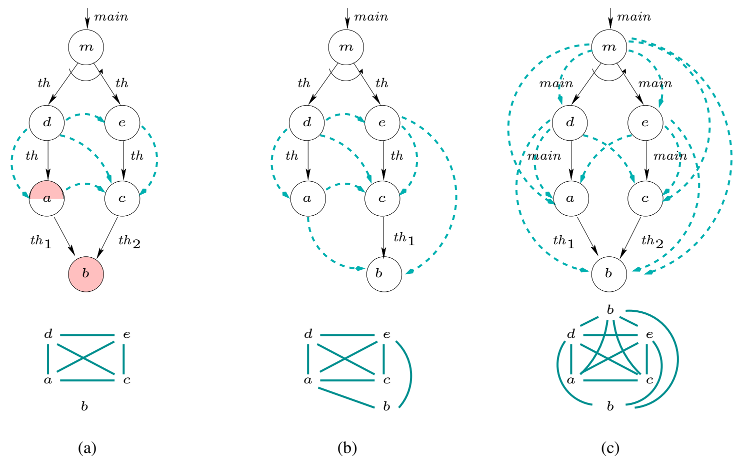
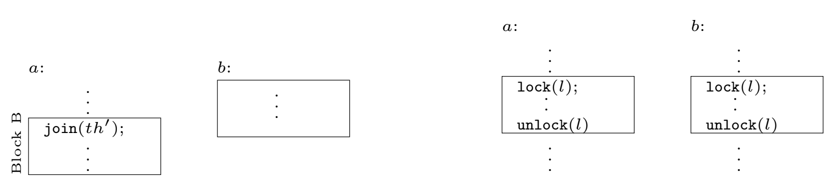



Fig. 8. Application of disjoint block Rules 1 and 2 on some example TPGs. Dashed arrows show the EB relation and thick undirected lines in the lower figures denote "disjoint-with".




(a) Disjoint Blocks in Rule 4 (b) Disjoint Blocks in Rule 4

Fig. 9. Rules 3 and 4

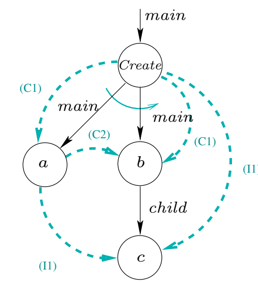
### 7.1 Race Detection Algorithm

The algorithm to detect races in EDP programs is shown in Algo. 2. We define the term "covers" used in the algorithm as follows. Let $P$ be an EDP program and let $B$ and $B'$ be two

```
Create:                 a:
1. p := 0;              10. x := p;
2. post(main, a);
3. post(main, b);

b:                      c:
20: y := p;             30. p := 10
21: child := create();
22: post(child, c);
```

(a) EDP program




(b) TPG annotated with EB relation

Fig. 10. Illustrating the race detection algorithm

blocks in $P$. We say that the pair of blocks $(B, B')$ *covers* the statements $s$ and $s'$ if either $s$ belongs to $B$ and $s'$ belongs to $B'$ or vice versa (see Sec. 6 for the definition of "belongs to").

**Algorithm 2:** Race Detection

**Data:** EDP Program $P$
**Result:** Set $PR$ of potential races
$PR := \emptyset$;
Find the list $CA$ of conflicting accesses in $P$;
**forall** *pair* $(s_1, s_2)$ *of conflicting accesses in* $CA$ **do**
  **if** *there are disjoint blocks* $B_1$ *and* $B_2$*, due to any of the rules, such that* $B_1$ *and* $B_2$ *covers* $s_1$ *and* $s_2$ **then**
    Declare $(s_1, s_2)$ to be non-racy;
  **else**
    Flag $(s_1, s_2)$ to be potentially racy;
    $PR := PR \cup \{(s_1, s_2)\}$;
  **end**
**end**

*Example.* We explain the application of Algo. 2 on a version of the example from Sec. 2, in Fig. 10a. A portion of the TPG, annotated with the executes before relation and the rules applied to derive them, is shown on the right. The abstract threads *main* and *child* are unique. The tasks accesses only one shared variable $p$ and the pairs of conflicting accesses are $\{(1,1),(1,10),(1,20),(1,30),(10,30),(20,30),(30,30)\}$. By Rule 1, tasks `Create`, $a$, and $b$ are pairwise disjoint, since they are always posted to the same unique abstract thread - *main*. Similarly, task $c$ is disjoint with itself. Hence the pairs $(1,1),(1,10),(1,20)$, and $(30,30)$ are declared to be non-racy by the algorithm. The unique task $b$ is the only task to post $c$. Hence Rule 2 applies and the block $B_1 = (20, \{20, 21, 22\})$ in task $b$ is disjoint with $c$. Thus the pair $(20,30)$ is declared to be non-racy by the algorithm. Since task `Create` executes before $c$, Rule 3 applies and the task `Create` is disjoint with $c$. Hence the pair $(1,30)$ is declared to be non-racy by the algorithm. Similar is the case with the pair $(10,30)$.

*Soundness of rules.* We can now prove the soundness of our disjoint block rules.

THEOREM 7.1. *The rules 1–4 are sound in that if any EDP program $P$ satisfies the premise of one of the rules, the identified blocks are indeed disjoint in $P$.*

Proof. The soundness of Rules 3 and 4 are standard.

To see that Rule 1 is sound, suppose tasks $a$ and $b$ in $P$ satisfy the conditions of the rule. Consider an execution $\rho$ of $P$ in which there is an instance of task $a$ and an instance of task $b$. Now there can only be one instance of $a$ in $\rho$ since $a$ is a unique task. If the instance of $b$ was posted by some other parent $c$ of $b$, then since $a$ executes before $c$, it must have finished execution before $b$ begins, and hence must be non-overlapping with $b$. On the other hand, if the instance of $b$ was posted by the (unique) instance of $a$, then clearly no part of the statements in the block $(ent_a, X)$ can overlap with statements of $b$. This completes the soundness of Rule 1.

The soundness of Rule 2 (Executes-Before) is immediate. □

## 8 ANDROID APPS AS EDP PROGRAMS

In this section we describe the structure and execution semantics of Android apps and show how we can view them as EDP programs.

An Android application (or *app*) is constituted using one or more of Android's four core components - *Activity*, *Service*, *Content Provider*, and *Broadcast Receiver*. An Activity is a component that provides a UI with which users can interact. An Activity undergoes a sequence of state transitions that permits it to interact with the user. These state transitions are triggered by lifecycle callbacks such as `onCreate`, `onStart`, `onResume`, `onPause`, `onStop`, `onRestart`, and `onDestroy`. These callbacks run on the *main* thread. The `ActivityManagerService`, a part of the Android system, controls the order in which the Activity callbacks are executed. Android also provides ways for executing background operations in threads other than the main thread. In this section, we model the Activity component of Android and the background processing.

*Modeling an Activity.* An Android application can be viewed as an event driven program with the Activity callbacks running as tasks on the *main* thread. The *sysTask* running on the *system* thread models `ActivityManagerService` and it controls the order of callbacks running on the *main* thread.

An Activity in our model is comprised of the tasks called `Preamble`, `Create`, `Resume`, `Pause`, `Stop`, `Restart`, and `Destroy`. The `Preamble` task does some initialization. The `Create` task comprise of instructions in the Android lifecycle callbacks `onCreate`, `onStart`, and `onResume`, in that sequence. The `Resume` task models `onResume` lifecycle callback, `Pause` task models `onPause`, `Stop` task models `onStop` callback, while `Restart` task consists of instructions in `onRestart`, `onStart`, and `onResume` callbacks, in sequence. `Destroy` task consists of `onDestroy` callback of the Activity. An EDP program can also have *UI tasks* that execute on the *main* thread. These tasks execute after `Create` or `Resume` tasks. The CFG of *sysTask* which controls the posting of task is shown in Fig. 11. It shows the order of posts of Activity life-cycle callbacks and one UI task. The EDP program for the running example in Fig. 1a is shown in Fig. 12.

Android provides *AsyncTask* feature that allows to run instructions in the background and allows reporting of results from the background thread to the *main* thread. We model an AsyncTask as having three tasks namely `doInBackground`, `onProgressUpdate` and `onPostExecute`. The `doInBackground` task which does background processing runs on a new thread while the task `onProgressUpdate` passes results of the background processing run on the *main* thread, and `onPostExecute`, which does clean up operations after the background processing finishes, run on the *main* thread.

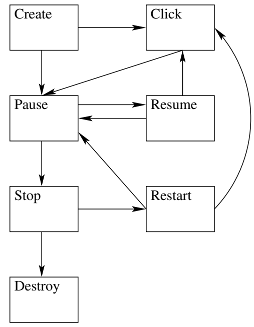



Fig. 11. Control flow of *sysTask*

```
sysTask:
 1.     post(main, Create)
 2. L1: post(main, Pause)
 3.     if (*) {
 4.       post(main, Resume)
 5.       goto L1
 6.     }
 7.     else {
 8.       post(main, Stop)
 9.       if (*) {
10.         post(main, Restart)
11.         goto L1
12.       }
13.       else {
14.         post(main, Destroy)
15.       }
16.      }
17.     }

m:
1. p := 0
2. system := create()
3. post(system, sysTask)

Create:
1. post(main, a)
2. post(main, b)

Pause:
1. skip

Resume:
1. skip

Stop:
1. skip

Restart:
1. skip

Destroy:
1. skip

a:
1. x := p

b:
1. y := p
2. child := create()
3. post(child, c)

c:
1. p := 10
```

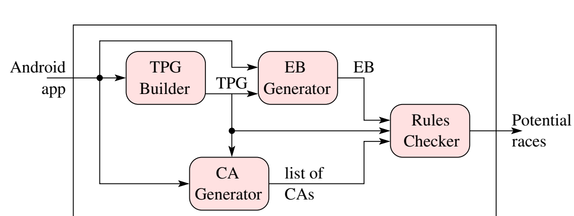
Fig. 12. Running example in Fig. 1a as an EDP program

## 9 IMPLEMENTATION

In this section we evaluate the recall of EB conditions (in Sec. 5) in computing the executes-before relation. We also assess the usefulness of EB rules in some downstream applications like race detection and redundant synchronization detection. We present the tool AndRacer, that statically analyzes Android applications for data races and also finds redundant synchronization blocks. We first describe the tool implementation followed by analyzing the result on 19 Android apps.

### 9.1 Tool Implementation

AndRacer takes an application package (as an *.apk* file) as input, and outputs a set of pairs of accesses that may be involved in a data race. A schematic representation of the AndRacer tool is shown in the figure alongside. The tool has four components: (1) the TPG Builder, to construct the TPG of the input app, (2) the EB Generator, to compute the task pairs that are executes-before related, (3) the CA Generator, to compute the list of conflicting access pairs, and (4)



the Rules Checker, to apply the rules on the conflicting access pairs to determine if they are racy or not.

*TPG Builder.* The TPG Builder relies on having an entry method for the application. The tool uses the FlowDroid framework (Arzt et al. 2014) to translate an Android application to one having an entry class, `DummyMain`, and an entry method, `dummyMain`. The `dummyMain` method posts all the life-cycle callbacks of the Android components.

We assume single run-time instance of a component. The callbacks of each component are independent and do not trigger the life-cycle callbacks of other components. Android has a special class, *Fragment* which has its own life-cycle callbacks. The callbacks of a Fragment are not causally related to the callbacks of the Android components and we consider Fragment as another component. The TPG Builder considers this special class, in addition to the four components (see Sec. 8), while constructing the TPG.

The TPG Builder first finds the nodes in the TPG which essentially are the tasks. In an Android application, callbacks correspond to tasks. The TPG Builder collects the callbacks using FlowDroid starting from the `dummyMain` method, which is the root node of the TPG, and the other callbacks are ones that are reachable from it via posts. Starting from the `dummyMain` method the tool analyzes the statements for the posts of the callbacks. The life-cycle callbacks have standard names like `onCreate`, `onStart`, etc., and can easily be identified. The application callbacks are posted using methods like `Handler.post`, `Thread.start`, `Timer.schedule`, `AsyncTask.execute`, etc. The post statement determines an edge from the task that has the post to the task being posted in the TPG. Each post edge has attributes for abstract thread for the post, uniqueness of the abstract thread, uniqueness of the post and the order of the post.

The abstract thread is identified using the points-to-set. The uniqueness of the abstract thread and post is decided by checking if the allocation-site/post statement is in a loop or in a task posted non-uniquely. Finally, the order of the post at a post statement is determined by the number of tasks that may-be-posted until a post instruction node in the CFG of the task.

*EB Generator.* The EB Generator component of AndRacer builds the executes-before relation between all possible pairs of callbacks in a given Android Application. We implemented algorithm Algo. 1 to soundly compute the executes-before relation based on the conditions in Fig. 4 and inferences in Fig. 6.

*CA Generator.* The CA Generator component of AndRacer collects the set of accesses to shared variables and marks whether they are read or write. For each callback pair that may be posted to different threads (which is inferred from the labels of incoming edges to the callbacks in the TPG) and for each pair of accesses in the callback pair, the CA Generator checks whether the pair of accesses conflict. If so, the access pair is marked as conflicting.

The tool uses the *points-to analysis* computed by the context and flow insensitive Spark framework (Lhoták and Hendren 2003), to decide on conflicting access the access pairs. We tried to incorporate other candidate points-to-analysis frameworks guaranteeing better precision. But, unfortunately the frameworks did not work as expected or were imprecise in specific scenarios. The imprecise points-to analysis by Spark leads to false conflicting accesses. In order to reduce the number of false conflicting accesses, we designed and implemented a simple *escape analysis* and object-sensitive *points-to analysis* for *this* pointers which are explained next.

Escape Analysis: Here we give a brief description of the analysis. The analysis determines whether an object allocated in a callback can ever be accessed outside the scope of the callback. Each callback, represented as an inter-procedural CFG, may have a set of allocation nodes (or abstract objects) with allocation sites in the callback inter-procedural CFG. We represent points-to information as a map from *access paths* to the sets of allocation nodes it can point to. An *access path* is a sequence of references. For example, an access path *f.g.h* is headed by a root allocation node pointed by $f$ and *g,h* are the other allocation nodes that may be reachable from *f*.

The idea of escape analysis is based on the following observation, an allocation node $o_n$ with an allocation site in the callback $c$ may escape the scope of the task, if any reference in the prefix of the access path $p$ to $o_n$ may-point to an allocation node with allocated site not in the callback.

This implies that an object created in some callback $c'$, may access the object allocated in the callback $c$. We implemented a conservative *may escape analysis*.

Object Sensitive *this*-pointer Analysis: The analysis is based on the observation that significant number of accesses in the applications conflict on the *this* reference of the JAVA class. AndRacer checks if the accesses are conflicting on the *this* reference and resolves the points-to set of *this* on demand. The base reference *this* is tracked at the caller. The points-to set of *this* is computed at the caller and the conflicting accesses are checked for intersection of points-to set.

*Rules Checker.* Given a list of conflicting access pairs in an Android app and the TPG for the app, the disjoint block rules described in Sec. 7 are applied to mark the conflicting access pairs that cannot execute in parallel in a callback pair. The Rules Checker component of the tool uses the executes-before information, from the EB Checker component, to implement the disjoint block rules.

We describe here the implementation of one of the rules - the "Lock" rule. One of the commonly used synchronization mechanisms in the apps is the use of *synchronized* blocks and *synchronized* methods. Hence our implementation considers only these mechanisms. We used context and flow insensitive points-to-set analysis to get allocation nodes corresponding to the object on which a lock held. In Java, every *synchronized* method/block keyword acquires a lock on some object, except in the case of synchronized `public static` methods. In that case, the lock is acquired on the `class` to which the `static` method belongs.

With these assumptions, we implemented a simple *lockset* algorithm to find the set of locks held at each statement in the `Jimple` representation of the input Android app. We define a *lock set* as a set of multi-set of objects (allocation nodes) or classes (in the case of static synchronized methods). Let $L_s$ be the lock set computed for a statement $s$. An element $M$ of $L_s$ is a multi-set that exactly contains the classes and the allocation nodes of objects on which the thread is holding the locks. Lock set for each statement is computed using data-flow analysis on the CFG of each task.

The computed lock sets are then used to check whether a pair of statements would have a common lock in every execution *i.e.* they may happen in parallel or not. Let $L_{s_1}$ be the lock set computed for a statement $s_1$ and $L_{s_2}$ be the lock set computed for statement $s_2$. Then statements $s_1$ and $s_2$ are cannot happen in parallel if $M_{s_1} \cap M_{s_2} \neq \phi : \forall M_{s_1} \in L_{s_1}, \forall M_{s_2} \in L_{s_2}$.

For example, consider tasks $a$ and $b$ in Fig. 13. Let the points-to-set of object $o_1$ be allocation node $a_1$ and the points-to-set of object $o_2$ be allocation nodes $a_1$ and $a_2$. Then the lock set associated with statement $s_1$ is $\{\{\},\{a_1\}\}$ and statement $s_2$ is $\{\{Hello, a_1\},\{Hello, a_2\}\}$.

```
class Hello {
  ...                                        |   synchronized public static void bar() {
  void foo() {                               |     synchronized(o2) { // s2 }
    // s1                                    |   }
  }                                          |
  Runnable a = new Runnable() {              |   Runnable b = new Runnable() {
    public void run() {                      |     public void run() { bar(); }
      foo();                                 |   }
      synchronized(o1) { foo(); }            |   public static void main(String[] args) {
    }                                        |     ... // calls leading to execution of a and b
  }                                          |   }
}
```

Fig. 13. Illustrating application of Rule 5

Consider the scenario where some thread is executing statement $s_1$ with a lock on $a_1$, now another thread can be executing statement $s_2$ with the locks held either on $Hello$ and $a_1$ or on $Hello$ and $a_2$. There is a lock common in $\{a_1\}$ and $\{Hello, a_1\}$ i.e. $a_1$. But there is no lock common in $\{a_1\}$ and $\{Hello, a_2\}$, in which case $s_1$ and $s_2$ may happen in parallel. Similarly we also need to check if there is a lock common in $\{\}$ and $\{Hello, a_1\}$ and also in $\{\}$ and $\{Hello, a_2\}$. If all such pairs have some lock in common, the statements cannot happen in parallel. In this case there are pairs that do not have any lock in common and hence the statements may happen in parallel.

*Redundant Synchronization.* We now describe another application of executes-before relation which is to detect the redundant synchronizations in an Android app. Apart from computing whether two statement are potentially racy, the "Lock" rule can also be used to identify *redundant synchronized blocks*. A *redundant synchronized block* is one which contains no such access that may happen in parallel with any other conflicting access even without it being synchronized. There is a performance overhead associated with entering and exiting a synchronized block. Hence getting rid of the redundant synchronized blocks improves the performance of the application.

We implemented a simple algorithm to detect redundant synchronizations as follows: First a set $S$ of all statements is computed which contain accesses that are marked as potentially racy after the application of disjoint block rules 1-3 (the rules other than the "Lock" rule). The CFG of every task is then traversed while maintaining a set *RSync* and stack *Stk*. *RSync* keeps the set of synchronized blocks that are redundant and *Stk* contains the synchronized blocks for whose start has been encountered but not its end. This is to account for nested synchronized blocks. If the statement encountered is a `lock`, it is pushed to *Stk* and is added to *RSync* as a representation for the corresponding synchronized block. If the statement encountered is an `unlock`, the corresponding `lock` is popped from *Stk*. If a statement is encountered that is in set $S$ of potentially racy access, then all the synchronized blocks present in the *Stk* are removed from *RSync*.

## 9.2 Benchmarks

We ran our tool on 19 Android applications to demonstrate the usefulness of the executes-before rules. We use latest versions of these well known real-world apps. Some of them are used in an earlier work relating to Android (Wu et al. 2019). Table 2 summarizes the features of the applications which are taken from various domains like finance, health, security, network, education, etc. Applications with multiple threads were selected for the experiments and the column "Thds", in the table, gives the number of threads in an application. The "Tasks" column gives the number of tasks. The smallest app analyzed, Child Monitor, has 1K

Table 2. Benchmark statistics

| App | LoC (K) | Thds | Tasks |
|---|---|---|---|
| Child Monitor | 1.0 | 3 | 34 |
| Aard2 | 4.9 | 7 | 88 |
| Dns66 | 4.9 | 7 | 47 |
| Character Recognition | 6.5 | 5 | 25 |
| A2DP Volume | 6.8 | 9 | 113 |
| AarogyaSetu | 8.2 | 3 | 61 |
| KeePassDroid | 18.1 | 5 | 79 |
| OpenApk | 2.1 | 5 | 34 |
| DeskCon | 3.1 | 13 | 64 |
| ClipStack | 3.9 | 5 | 146 |
| Crescent Cash | 5.3 | 13 | 165 |
| BitCoinium Prime | 7.0 | 13 | 115 |
| OSMonitor | 14.2 | 4 | 68 |
| AnyMemo | 23.3 | 14 | 251 |
| Mileage | 44.5 | 12 | 109 |
| AntennaPod | 54.5 | 11 | 458 |
| OwnCloud | 56.0 | 14 | 390 |
| k9mail | 76.1 | 6 | 296 |
| Fbreader | 76.5 | 20 | 285 |

lines of Java code (excluding comments and blank lines), as indicated by the "LoC" column, while the largest of them all, Fbreader, has 76.5K LoC.

### 9.3 Results

We conducted the experiments on an Intel Xeon W-2295 CPU with 256GB RAM running Ubuntu 20.04 LTS. Table 3 shows the recall of executes-before (EB) conditions (proposed in Sec. 5) in computing the executes-before relation, when we ran AndRacer on the apps. The "Tool EB" column gives the number of executes-before pairs of callbacks that are computed by the tool while the "Man. EB" column gives the number of executes-before pairs, that we found out on manual inspection. The manual inspection is done on a subset of Android components, mostly the *Activity* component. The ratio of the pair of callbacks reported by AndRacer to the pair of callbacks reported manually is given in "Recall%" column. The "Time" column gives the time taken (in seconds) to compute the executes-before pairs.

Our tool performed well with a high recall value of 97%. The recall results demonstrates that our executes-before conditions were reasonably comprehensive that they fit the natural patterns of executes before scenarios found in these apps. The tools missed EB pairs mostly due to the imprecise flow sensitive analysis of Spark framework and that FlowDroid did not report some callbacks, hence execute-before pairs involving these callbacks have been left out.

Table 4 gives statistics on the potential races detected by the tool. The "CA" column gives the number of conflicting accesses detected. The table is further structured to give data on three main features of the tool, we intended to evaluate. The "EB usefulness" part gives data to indicate the effectiveness of the EB relations in reporting CA pairs as non-racy.

Table 3. Executes-before pairs reported by AndRacer

| App | Tool EB | Man. EB | Recall% | Time (s) |
|---|---|---|---|---|
| Child Monitor | 44 | 45 | 97.7 | 0.05 |
| Aard2 | 74 | 74 | 100.0 | 0.31 |
| Dns66 | 22 | 22 | 100.0 | 0.10 |
| Character Recognition | 15 | 15 | 100.0 | 0.05 |
| A2DP Volume | 148 | 158 | 94.0 | 0.42 |
| AarogyaSetu | 82 | 38 | 100.0 | 0.11 |
| KeePassDroid | 77 | 83 | 93.0 | 0.21 |
| OpenApk | 25 | 28 | 89.0 | 0.05 |
| DeskCon | 71 | 37 | 100.0 | 0.11 |
| ClipStack | 119 | 40 | 100.0 | 2.14 |
| Crescent Cash | 130 | 42 | 100.0 | 0.91 |
| BitCoinium Prime | 38 | 22 | 100.0 | 0.20 |
| OSMonitor | 14 | 14 | 100.0 | 0.03 |
| AnyMemo | 175 | 81 | 95.0 | 0.52 |
| Mileage | 54 | 58 | 93.0 | 0.18 |
| AntennaPod | 310 | 151 | 97.3 | 3.47 |
| OwnCloud | 222 | 136 | 100.0 | 1.13 |
| k9mail | 138 | 20 | 100.0. | 1.11 |
| Fbreader | 344 | 120 | 97.5 | 2.85 |

The "Race statistics" figures indicate the races reported and the precision in detecting actual races while the "Redn. Sync." figures indicate the usefulness in detecting redundant synchronizations. "Syn" and "EB" columns give the number of CA pairs eliminated, as non-racy, due to the use of synchronizations and execute-before relation, respectively. Note that, some pairs can be eliminated by both. The "SoleEB" column gives the number of CA pairs eliminated solely due to executes-before relation. "SoleEB%" column gives the percentage of CA pairs eliminated solely due to executes-before relation. Moving on to race statistics, the "PR" column gives the number of CA pairs flagged as potentially racy by the tool and "AR" is a conservative count of actual races found on manual inspection. Due to the complex control flow, we not able to inspect some of the apps for actual races. The top section of the table gives AR values for those apps which we could manually analyze. The percentage of actual races in the potential races flagged by the tool, as a measure of precision, is given under the "Prec%" column. The time taken to report races is given under the "Time" column. Recall that our tool also reports the use of redundant synchronization. The "RSB" column gives the number of redundant synchronizations detected by the tool and the number in parenthesis is the actual count of synchronizations used in the apps. The "Time" column here gives the time taken to report the redundant synchronization count.

*Discussion.* We note that our tool is able to filter out a large part of the conflicting critical access pairs as non-racy (on the average of 45.3% of CAs are eliminated). The proposed EB based rules were found to be useful in eliminating CA pairs as non-racy. On an average, 57% of CA pairs were eliminated due to the use of EB rules. It is worthwhile to note that the EB rules were the sole factor in eliminating CA pairs in the Crescent Cash app which had well over 6000 CAs. The figures for Character Recognition app are similarly encouraging.

Our tool is fairly precise in that it reported fewer false positives. We were able to manually inspect some of the apps for actual data races. Some of them we could not inspect due to their complex control flow. Based on the ones we inspected, out tool is precise with an average of 61%.

Table 4. Data Races reported by AndRacer

| App | CA | EB usefulness | | | | Race statistics | | | | Redn. Sync. | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Syn | EB | SoleEB | SoleEB% | PR | AR | Prec% | Time(s) | RSB | Time(s) |
| Child Monitor | 22 | 0 | 10 | 10 | 100.0 | 12 | 12 | 100 | 2.3 | 0 (/0) | 2.2 |
| Aard2 | 31 | 5 | 6 | 6 | 54.5 | 20 | 6 | 30 | 21.1 | 4 (/6) | 21.1 |
| Dns66 | 51 | 21 | 22 | 3 | 12.5 | 27 | 0 | 0 | 11.4 | 5 (/7) | 11.4 |
| Character Recognition | 43 | 0 | 30 | 30 | 100.0 | 13 | 13 | 100 | 2.7 | 0 (/0) | 2.7 |
| A2DP Volume | 47 | 0 | 17 | 17 | 100.0 | 30 | 30 | 100 | 7.6 | 0 (/0) | 7.3 |
| AarogyaSetu | 8 | 0 | 2 | 2 | 100.0 | 6 | 6 | 100 | 70.0 | 0 (/0) | 69.0 |
| KeePassDroid | 49 | 45 | 14 | 2 | 4.3 | 2 | 0 | 0 | 20.0 | 0 (/1) | 19.9 |
| OpenApk | 693 | 0 | 148 | 148 | 100.0 | 545 | | | 9.1 | 0 (/0) | 9.0 |
| DeskCon | 122 | 0 | 29 | 29 | 100.0 | 93 | | | 7.5 | 0 (/0) | 7.5 |
| ClipStack | 371 | 269 | 1 | 1 | 0.4 | 101 | | | 10.6 | 4 (/17) | 10.7 |
| Crescent Cash | 6058 | 0 | 5794 | 5794 | 100.0 | 264 | | | 72.5 | 0 (/0) | 70.6 |
| BitCoinium Prime | 156 | 0 | 57 | 57 | 100.0 | 99 | | | 17.7 | 0 (/0) | 17.5 |
| OSMonitor | 3911 | 0 | 221 | 221 | 100.0 | 3690 | | | 4.8 | 0 (/0) | 4.9 |
| AnyMemo | 3602 | 98 | 96 | 96 | 49.5 | 3408 | | | 27.6 | 3 (/5) | 27.4 |
| Mileage | 2592 | 225 | 951 | 909 | 80.2 | 1458 | | | 5.5 | 0 (/1) | 5.7 |
| AntennaPod | 2193 | 338 | 383 | 383 | 53.1 | 1472 | | | 157.8 | 50 (/57) | 148.3 |
| OwnCloud | 6130 | 15 | 73 | 62 | 80.5 | 6053 | | | 78.9 | 5 (/6) | 77.5 |
| k9mail | 5146 | 4948 | 2 | 2 | 0.04 | 196 | | | 97.9 | 26 (/36) | 97.8 |
| Fbreader | 226 | 0 | 156 | 156 | 100.0 | 70 | | | 29.3 | 44 (/50) | 29.2 |

*False Positives.* One of the reasons for imprecision in race detection is due to the Spark points-to set analysis. Our tool considers that multiple instances of a task is represented by one "abstract" task. There are several scenarios in the apps where multiple components post a task. Hence our tool loses precision, since none of the rules apply, accounting for some false positives. For example, in Dns66 app, during the initialisation of the *NotificationBuilder* field in the `onNewIntent` callback of MainActivity, it can get updated in `doInBackground` task running in a different thread. The two accesses are deemed potentially racy but even if they may run in parallel, fields of different instances of the task will be accessed. Our tool misses out on this pair because another instance of the `doInBackground` task is created by another component, hence violating our assumption.

The redundant synchronizations analysis, detects use of synchronization constructs in the applications. The tool found that some of the apps like Aard2, Dns66, AntennaPod, OwnCloud, and Fbreader relied on a lot of synchronizations which were not needed since their shared accesses do not happen in parallel (as detected by the EB conditions).

To summarize, our tool performed well in detecting data races and redundant synchronizations, despite the use of imprecise points-to analysis. The proposed executes-before conditions played a significant role in the performance numbers of the tool.

## 10 RELATED WORK

We group related work according to work on executes-before, MHP, and dynamic and bounded model-checking based techniques for EDP programs, and discuss our work in relation to them.

*Executes-Before analysis.* Both Hu and Neamtiu (2018) and (Wu et al. 2019) consider the problem of statically determining executes-before ("happens-before" in their terminology) pairs as part of their goal of statically detecting *event-based* races in Android apps. Event-races are conflicting accesses that are not causally ordered in the application (for instance, we would like an access to happen *after* the initialization and a free to happen *after* an access). They build a happens-before graph out of different components (including tasks and click events) in the app, with edges $A \prec B$ meaning that $A$ "happens-before" $B$. They then use this graph to order conflicting accesses and report the ones not ordered as potential event races. To begin with, event races are a weaker notion than the standard races we target in this paper: for instance, two accesses that are well-synchronized by locks (and hence non-racy) may be declared as event races simply because there is no fixed order between them. Secondly, while their happens-before rules are similar in spirit to our executes-before conditions, their rules are *not* sound for general EDP programs. In particular, one of the rules in Hu and Neamtiu (2018) says that if $A_1 \prec A_2$ and $A_1$ posts $A_3$ and $A_2$ posts $A_4$, then we can infer that $A_3 \prec A_4$; which is clearly unsound unless we assume $A_3$ and $A_4$ are posted to the *same unique* thread. Similarly, in Wu et al. (2019) the rule Intra-Post says that if event $(e_x, c_x)$ (i.e. task $e_x$ in post-context $c_x$) posts $(e_y, c_y)$ (all on and to the same thread $t$), then $(e_x, c_x) \prec (e_y, c_y)$. If we consider the program alongside, this rule is easily seen to be unsound. Thus the aim of the rules in these two works is to be able to produce a small set of potential event races with a low false positive rate, with no intention of being sound. In contrast, we want our execute-before rules to be sound, given the downstream applications of MHP analysis, (sound) data race and redundant synchronization detection, and data-flow analysis.

```
mtask:

while (*) {
  t := create();
  post(t,mtask);
}
```

*MHP Analysis.* Kahlon et al. (2009) give a static analysis to detect races in multi-threaded C programs with asynchronous function calls which is similar to EDP programs. Their main focus is on doing a context-sensitive points-to and must-held lockset analysis for C programs in the presence of function pointers. The MHP rules they give essentially correspond to standard fork-join and lock-unlock rules. They also exploit statement ordering within a thread but appear weaker than our EB-based rules. In particular they would not be able to detect disjointness (or "not MHP") of tasks $a$ and $b$ in Fig 8, where we add a post of $b$ from task $c$ to a new thread $th'$.

The algorithm by Albert et al. (2015) computes precise MHP information for fork-join asynchronous programs. This is not very useful in our setting (for example in Android apps) where joins appear to be rarely used.

Since Android apps are Java-based, one may ask if static race-detection techniques for Java could be used for Android apps. While many of the techniques for obtaining a precise set of conflicting accesses (for example Naik et al. (2006)) would help here too, the MHP analysis would not be sound as they do not consider the task posting feature of EDP programs. Moreover, these techniques typically drop soundness in favour of precision. For instance, Naik et al. (2006) declare statements to be non-MHP even if two may-held locks may-alias. The nADROID tool of Fu et al. (2018) tries to address task posting in Android apps by converting them to a standard Java program in which each callback is on a *different* thread, and then invoking a Java race detector like CHORD (Naik et al. 2006). However, as one would expect, this approach leads to a lot of false positives.

*Dynamic and bounded analysis.* One of the early works on dynamic event-based race detection is based on the idea of *race coverage* (Raychev et al. 2013) which eliminates races on conflicting accesses that are used for synchronization. The algorithm is implemented in the tool called EventRacer and detects races in web applications with high precision. Hsiao et al. (2014) developed the race detection tool called CAFA which is based on the idea of non-commutativity of events. The tool first detects concurrent events based on a causality model which defines relations between events due to the operations on the event queue and the properties of the events. This permits a more precise definition of happens-before relations. The model permits multiple threads and various Android components, but focuses on use-after-free races. Bielik et al. (2015) address the issue of scalability of dynamic analysis techniques through the use of an efficient algorithm for building and querying the happens-before graph. The work focuses on data races between events handled by a single main thread. In general, dynamic analysis techniques are inherently unsound in that they may have false negatives, while we aim for soundness.

DroidRacer (Maiya et al. 2014) use a bounded model-checking approach to detect a wide range of event-based races. The authors give a formal semantics of event- driven systems that consider both thread interleavings and event dispatch. In another bounded model-checking approach (Majumdar and Wang 2015) implement a *phase-bounding* algorithm, to analyze C programs that have an execution model which supports asynchronous programming, in a tool called BBS. The model assumes an application to have a single worker thread process with a queue to which tasks can be posted. BBS implements a sequentialization algorithm which replaces asynchronous posts with "normal" function calls. The resulting sequential program is fed into the bounded model checker CBMC. While such approaches can be expected to be very precise, they are not scalable and are inherently unsound.

Emmi et al. (2015) prove the decidability of program analysis in EDP programs through a reduction from the control-state reachability problem for asynchronous programs to one in Petri Data Nets. The model puts a bound on the number of tasks and buffers, with the buffers being unordered and tasks making non-recursive procedure calls. State explosion is an issue for model-checking based approaches and it would be difficult to get them to scale to the size of apps analyzed in our work.

## 11 CONCLUSION

In this paper we have given a sound and efficient technique, with good recall, to statically identify executes-before pairs in event driven programs. The executes-before information is shown to be effective in downstream analyses like data race detection and identifying redundant synchronization blocks in Android apps.

In future work we would like to explore the use of the executes-before information in sound detection of event-based races, as well as in efficient and precise data-flow analysis for event driven programs.

## REFERENCES


Elvira Albert, Samir Genaim, and Pablo Gordillo. 2015. May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization. In *Proceedings of the Static Analysis - 22nd International Symposium, SAS 2015 (Lecture Notes in Computer Science, Vol. 9291)*. Springer, Saint-Malo, France, 72–89.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, Edinburgh, United Kingdom, 259269.

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2015. Scalable race detection for Android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015*. ACM, Pittsburgh, PA, USA, 332–348.

Nikita Chopra, Rekha Pai, and Deepak D'Souza. 2019. Data Races and Static Analysis for Interrupt-Driven Kernels. In *Proceedings of the 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019 (Lecture Notes in Computer Science, Vol. 11423)*. Springer, Prague, Czech Republic, 697–723.

Arnab De, Deepak D'Souza, and Rupesh Nasre. 2011. Dataflow Analysis for Datarace-Free Programs. In *Proceedings of the Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011 (Lecture Notes in Computer Science, Vol. 6602)*, Gilles Barthe (Ed.). Springer, Saarbrücken, Germany, 196–215. https://doi.org/10.1007/978-3-642-19718-5_11

Michael Emmi, Pierre Ganty, Rupak Majumdar, and Fernando Rosa-Velardo. 2015. Analysis of Asynchronous Programs with Event-Based Synchronization. In *Proceedings of the 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015 (Lecture Notes in Computer Science, Vol. 9032)*. Springer, London, UK, 535–559.

Dawson R. Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003*, Michael L. Scott and Larry L. Peterson (Eds.). ACM, Bolton Landing, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

Xinwei Fu, Dongyoon Lee, and Changhee Jung. 2018. nAdroid: statically detecting ordering violations in Android applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018*, Jens Knoop, Markus Schordan, Teresa Johnson, and Michael F. P. O'Boyle (Eds.). ACM, Vösendorf / Vienna, Austria,, 62–74. https://doi.org/10.1145/3168829

Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. 2007. Thread-modular shape analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 266–277. https://doi.org/10.1145/1250734.1250765

Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. 2014. Race detection for event-driven mobile applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*. ACM, Edinburgh, United Kingdom, 326–336.

Yongjian Hu and Iulian Neamtiu. 2018. Static Detection of Event-based Races in Android Apps. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018*. ACM, Williamsburg, VA, USA, 257–270.

Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009*. ACM, Amsterdam, The Netherlands, 13–22.

Ondrej Lhoták and Laurie J. Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003*. Springer, Warsaw, Poland, 153–169. https://doi.org/10.1007/3-540-36579-6_12

Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*. ACM, Edinburgh, United Kingdom, 316–325.

Rupak Majumdar and Zilong Wang. 2015. Bbs: A Phase-Bounded Model Checker for Asynchronous Programs. In *Proceedings, Part I, of Computer Aided Verification - 27th International Conference, CAV 2015 (Lecture Notes in Computer Science, Vol. 9206)*. Springer, San Francisco, CA, USA, 496–503.

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, Ottawa, Ontario, Canada, 308–319. https://doi.org/10.1145/1133981.1134018

Veselin Raychev, Martin T. Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH*. ACM, Indianapolis, IN, USA, 151–166.

Abhishek Singh, Rekha Pai, Deepak D'Souza, and Meenakshi D'Souza. 2019. Static Analysis for Detecting High-Level Races in RTOS Kernels. In *Proceedings of the Formal Methods - The Next 30 Years - Third World Congress, FM 2019 (Lecture Notes in Computer Science, Vol. 11800)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer, Porto, Portugal, 337–353. https://doi.org/10.1007/978-3-030-30942-8_21

Nicholas Sterling. 1993. WARLOCK - A Static Data Race Analysis Tool. In *USENIX Winter*.

Diyu Wu, Jie Liu, Yulei Sui, Shiping Chen, and Jingling Xue. 2019. Precise Static Happens-Before Analysis for Detecting UAF Order Violations in Android. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019*. IEEE, Xi'an, China, 276–287.