Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency

Ahmed Bouajjani[1], Michael Emmi[2], Constantin Enea[1], Burcu Kulahcioglu Ozkan[3](✉), and Serdar Tasiran[3]

[1] Université Paris Diderot, Paris, France
[2] Nokia Bell Labs, Murray Hill, NJ, USA
[3] Koç University, Istanbul, Turkey
bkulahcioglu@ku.edu.tr

Abstract. We define a correctness criterion, called *robustness against concurrency*, for a class of event-driven asynchronous programs that are at the basis of modern UI frameworks in Android, iOS, and Javascript. A program is robust when all possible behaviors admitted by the program under arbitrary procedure and event interleavings are admitted even if asynchronous procedures (respectively, events) are assumed to execute serially, one after the other, accessing shared memory in isolation. We characterize robustness as a conjunction of two correctness criteria: *event-serializability* (i.e., events can be seen as atomic) and *event-determinism* (executions within each event are insensitive to the interleavings between concurrent tasks dynamically spawned by the event). Then, we provide efficient algorithms for checking these two criteria based on polynomial reductions to reachability problems in sequential programs. This result is surprising because it allows to avoid explicit handling of all concurrent executions in the analysis, which leads to an important gain in complexity. We demonstrate via case studies on Android apps that the typical mistakes programmers make are captured as robustness violations, and that violations can be detected efficiently using our approach.

1 Introduction

Asynchronous event-driven programming is a widely adopted style for building responsive and efficient software. It allows programmers to use asynchronous procedure calls that are stored for later executions, in contrast with synchronous procedure calls that must be executed immediately. Asynchronous calls are essential for event-driven programming where they correspond to callbacks handling the occurrences of external events. In particular, modern user interface (UI) frameworks in Android, iOS, and Javascript, are instances of asynchronous event-driven programming. These frameworks dedicate a distinguished main thread,

This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 678177).

called UI thread, to handling user interface events. Since responsiveness to user events is a key concern, common practice is to let the UI thread perform only short-running work in response to each event, delegating to asynchronous tasks the more computationally demanding part of the work. These asynchronous tasks are in general executed in parallel on different background threads, depending on the computational resources available on the execution platform.

The apparent simplicity of UI programming models is somewhat deceptive. The difficulty of writing safe programs given the concurrency of the underlying execution platform is still all there. A formal programming abstraction that is simple, yet exposes both the potential benefits and the dangers of the UI frameworks would go a long way in simplifying the job of programmers. Programs written against this abstraction would then be insensitive to implementation and platform changes (e.g., automatic load balancing). Indeed, the choice of parameters such as the number of possible threads running in parallel, the dispatching policy of pending tasks over these threads, the scheduling policy for executing shared-memory concurrent tasks, etc., should be transparent to programmers, and the semantics of a program should be independent from this choice. Therefore the conformance to this abstraction (i.e., a program can be soundly abstracted according it) would be a highly desirable correctness criterion.

The objectives of our work are (1) to provide such a programming abstraction that leads to a suitable correctness criterion for event-driven shared memory asynchronous programs, and (2) to provide efficient algorithms for verifying that a program is correct w.r.t. this criterion.

The programming abstraction we consider compares two semantics, the *multi-thread* and the *single-thread* semantics:

- The multi-thread semantics reflects the concurrency of the actual program: The main (UI) thread and asynchronous tasks posted to background threads interact over the shared memory in a concurrent way. No limit on the number of tasks, no limit on the number of threads, and no restriction on the dispatching and scheduling policies are assumed.
- The single-thread semantics is a reference model where a program is supposed to run on a single thread handling user events in a serial manner, one after the other. Each event is handled by executing its corresponding code including the created asynchronous tasks until completion. The asynchronous tasks created by an event handler (and recursively, by its callee) are executed asynchronously (once the execution of the creator finishes) serially and in the order of their invocation.

While the multi-thread semantics provides greater performance and responsiveness, the single-thread semantics is simpler to apprehend. The inherent nondeterminism due to concurrency and asynchronous task dispatching from the multi-thread semantics is not present in the context of the single-thread one.

We consider that a desirable property of a program is that its multi-thread semantics is a *refinement* of its single-thread semantics in the sense that the sets of observable reachable states of the program w.r.t. both semantics are exactly the same. A program that satisfies this refinement condition is said to be *robust against concurrency* (or simply *robust*). In fact, robustness violations correspond to “concurrency bugs”, i.e., violations that are due to parallelization of tasks, and that do not show up when tasks are executed in a serial manner.

Then, let us focus now on the problem of verifying the robustness of a given program. We show in this paper that, surprisingly, for the class of UI event-driven asynchronous programs, this problem can be reduced in linear time to the state reachability problem in *sequential programs*. This means that the robustness of such a concurrent program can be checked in polynomial time on an (instrumented) *sequential* version of the program, without exploring all its concurrent executions. Let us describe the way we achieve that.

First, we show that robustness against concurrency can be characterized as the conjunction of *event-serializability* and *event determinism*, which are variants of the classical notions of serializability and determinism, adapted to our context. Intuitively, since the single-thread semantics defines a unique execution, given a set of external events (partially ordered w.r.t. some causality relation imposed by the environment), then (1) the executions of the event handlers must be serializable (to an order compatible with their causality relation), i.e., the execution of each event handler and its subtasks can be seen as an atomic transaction, and (2) the execution of each event handler is deterministic, i.e., it always leads to the same state, for any possible scheduling of its parallel subtasks.

To search efficiently for event-serializability and event-determinism violations, we make use of *conflict-based approximations* in the style of [27], called conflict-serializability and conflict-determinism, respectively. Indeed, these conflict-based criteria do not take into account actual data values, but rather syntactical dependencies between operations (e.g., writing to the same variable), which makes them stronger, but also “easier” to check, while still accurate enough for catching real bugs, introducing rarely false positives, as our experiments show. We reduce verifying conflict event-serializability and conflict event-determinism to detecting cycles in appropriately defined dependency (or happen-before) relations between concurrent events and asynchronous procedure invocations, respectively. Our key contribution is that these cycle detections can be done by reasoning about the computations of *sequential programs* instead of concurrent programs, avoiding explicit encodings of (potentially unbounded) sets of pending tasks and exploring all their possible interleavings. Let us explain this in more details.

An event handler is conflict-deterministic when all its executions have conflict-preserving permutations where tasks are executed serially in the same order as in the single-thread semantics. Scheduling tasks in this order corresponds to the DFS (Depth First Search) traversal of the call-tree of tasks (representing the relation caller-callee). We show that detecting a conflict-determinism violation, i.e., an asynchronous execution with no serial DFS counterpart, can be done by reasoning about an instrumented version of the procedural program obtained from the code of the event handler by roughly, turning pasynchronous calls to synchronous ones. This instrumented program simulates *borderline violations*, if any, i.e., violations where removing the last action leads to a correct execution. We show that the amount of auxiliary memory needed to witness such violations is finite (and small). Moreover, such violations are "almost" asynchronous executions where tasks are scheduled serially according to the DFS traversal of the call-tree. Such executions can be simulated using synchronous procedure calls because roughly, the latter are also initiated according to the DFS traversal of the call-tree. However, they are interleaved in a different way compared to the asynchronous calls and the event handler must undergo a syntactic transformation described in Sect. 6.3.

As for conflict-serializability, a first issue in checking it is that event handlers may consist of different concurrently-executing tasks. This issue is solved by assuming that the conflict-determinism check is done a-priori. If this check fails then the program is not robust and otherwise, checking conflict-serializability can assume sequential event handlers which are in fact the instrumented procedural programs used in the conflict-determinism check.

Even assuming sequential event handlers, general results about conflict-serializability state that this problem is PSPACE-complete for a fixed number of threads [6,15], and EXPSPACE-complete for an unbounded number of threads [10] (assuming a fixed data domain and absence of recursive procedure calls). However, we prove that, in the programming model we consider in this paper, the problem of checking conflict-serializability is *polynomial*! This result relies on two facts: (1) there is only one distinguished thread, the UI thread, for which the order in which procedure invocations are executed is relevant, and (2) we assume that each asynchronous task executed in the background (not on the UI thread) is running on a fresh thread. This assumption is valid since background threads are not manipulated explicitly by the programmer but by the runtime, and therefore, we need to consider the situation where concurrency is maximal.

In fact, we show that when events are conflict-deterministic, the problem of checking conflict-serializability can also be reduced to a reachability problem in a sequential program. Again, we prove that it is sufficient to focus on a particular class of (borderline) violations of conflict-serializability. Then, we show that detecting these violations can be done by reasoning about the executions of a program where events are executed in a sequential manner, in any order (chosen nondeterministically), and where the tasks generated by each event are executed as in the single-thread semantics. For that, we define an instrumentation of that program that consists in simulating the delaying effects of the multi-thread semantics, guessing the actions involved in the violation and tracking the dependencies between them in order to check the correctness of the guess (that they indeed form a cycle). The cycle detection in the case of conflict-serializability is technically more complex than in the case of conflict-determinism. But still, a crucial point in the reduction is that we do not need to store the whole cycle during the search, but it is enough to maintain a fixed number of variables to traverse the elements of this cycle. This leads to a polynomial reduction of the conflict-robustness problem to a reachability problem in a sequential program.

Our reductions hold regardless of the used data domain, for programs with recursive procedure calls, and unbounded numbers of events and tasks. These reductions allow to leverage existing analysis tools for sequential programs to check conflict-robustness. When the data domain is bounded, we obtain a polynomial-time algorithm for checking conflict-robustness for UI event-driven asynchronous programs (with recursive procedure calls, and unboundedly many events and tasks).

We validate our approach on a set of real-life applications, showing that with few exceptions all detected robustness violations are undesirable behaviours. Interestingly, the use of conflict versions of the correctness criteria characterizing robustness is efficient and quite accurate, producing only few false positives (that can be eliminated easily).

Finally, let us mention that our work also leads to an efficient approach for verifying functional correctness of UI event-driven asynchronous programs that consists in reducing this problem to two separate problems: (1) showing that the program is functionally correct w.r.t the single-thread semantics, and (2) showing that it is robust against concurrency. Both of these problems can indeed be solved efficiently by considering only particular types of computations that are captured by sequential programs.

To summarize, our contributions are:

– Introduction of the notion of robustness against concurrency that provides a programming abstraction for event-driven asynchronous programs, and its characterization as the conjunction of event-serializability and event-determinism.
– Efficient algorithms for checking robustness based on reductions from conflict event-serializability and conflict event-determinism to state reachability problems in sequential programs. Decidability and complexity results for verifying robustness in the case of finite data domains.
– Experimentations showing the relevance of our correctness criteria and the efficiency of our approach.

2 Motivating Examples

We demonstrate the relevance of robustness using several excerpts from Android applications. To argue that robustness is not too strong as a requirement, we discuss two concurrency bugs reported in open-source repositories that are also robustness violations, more precisely, event-serializability and event-determinism violations. We also provide a typical example of a robust program.

2.1 A Violation to Event Serializability

```
ActionEditText msgTxt;
boolean onKey(...) {
  // actions on UI thread
  new SendTask().execute();
}

void onDoubleClicked(String name){
  text += "␣" + name;
  msgTxt.setText(text);
}

class SendTask extends AsyncTask {
void onPreExecute(){
  e.command = msgTxt.getText();
}
void doInBackground(..) {
 write msgTxt.getText() into JSON obj
 /* corrected version:
 write e.command into JSON obj */
}
}
```

Fig. 1. A program with an event-serializability violation.

Figure 1 lists a real code excerpt from the Android IrcCloud app [2] for chatting on the IRC. Under the concrete multi-thread semantics, the user event of pressing the "send" key is handled by the procedure `onKey`. Actions associated with this event handler include actions performed by `onKey` on the main (UI) thread, actions performed by `SendTask.onPreExecute()` on the UI thread before the actions performed asynchronously on a background thread by `SendTasks`'s `doInBackground` procedure. Another event handler in this example is `onDoubleClicked`, which appends to the message text the name `name` of the user whose name is clicked on. The multi-thread semantics allows interference between the two event handlers, `onDoubleClicked` can interleave with `doInBackground`. In contrast, the single-thread semantics allows no such interference. The event handlers and the asynchronous tasks they create are executed entirely on the UI thread, and all the tasks created by `onKey` are executed before any other event handler invocation.

This program is not robust and a violation can be generated under the following scenario. Suppose that the user types "Hello", presses "send", and then double-clicks on another IRC user's name. Under the multi-thread semantics, `onDoubleClicked` may start running on the UI thread while `SendTask.doInBackground` is in progress. These two procedures' accesses can interfere with each other. In particular, the ordering of `msgTxt.getText()` with respect to the appending of `name` to `msgTxt` determines whether "Hello" or "Hello foo" gets sent on the network. Moreover, since `onKey` first records `msg.getText()` to a field `e.command`, an execution of these two events can end in a program state in which `e.command` contains "Hello" while `msgTxt` contains "Hello foo". This end state is not possible with *any* execution of these two event handlers under the single-thread semantics, where the event handlers are executed serially one after the other. This is a violation to event serializability. Actually, this behavior was reported as a bug, and the code was updated [1] so that `e.command` (instead of `msgTxt`, which may have changed) is written into a JSON object and sent on the network. It was the designers' intent for the entire event handling code for the "send" key to appear atomic. With this modification the program becomes robust.

2.2 A Violation to Event Determinism

```
void iconPackUpdated(){
 new Thread( new Runnable(){
  void run() {
    ..
   mAdapter=new AppRowAdapter(..);
  } } ).start();
 new Handler().postDelayed(
  new Runnable() {
   void run() {
    ..
    if(setAdapter)
     listView.setAdapter(mAdapter);
    ..
    mAdapter.notifyDataSetChanged();
   } }, 1000);
}
```

Fig. 2. A program with an event-determinism violation.

Figure 2 lists an event handler called `iconPackUpdated` which creates an asynchronous task (the first runnable to be executed by the created thread) to initialize the `mAdapter` object. Then it creates another task, to be run by the UI thread, that uses `mAdapter` to update the list view of displayed icons. In an effort to ensure that the second task runs after the first task completes, the programmer posts the second task after a second's delay.

Under the concrete multi-thread semantics, it is possible for the first task not to complete even after a second. In this case, the second runnable code will produce a null pointer exception, while in other schedules, the code works as intended. Although the programmer had intended a deterministic outcome there are executions with different outcomes, including errors. Therefore, this event handler is not event-deterministic, and not robust.

2.3 A Robust Program

The program in Fig. 3 has two event handlers `searchForNews` and `showDetail` which can be invoked by the user to search for news containing a keyword and to display the details of a selected news respectively.

The procedure `searchForNews` creates two `AsyncTask` objects `SearchTask` and `SaveTask` whose `execute` method will invoke asynchronously `doInBackground` followed by `onPostExecute`, in the case of the former. Under the multi-thread semantics, `doInBackground` is invoked on a new thread and `onPostExecute` is invoked

```
// Event 1
void searchForNews(String key) {
  new SearchTask.execute(key);
  new SaveTask.execute(key); }

// Event 2
void showDetail(int id) {
  // show detail of the idth news
  new DownloadTask.execute(id); }

class SaveTask extends AsyncTask {
  void doInBackground(String key) {
    // write key to the database } }
```

```
class SearchTask extends AsyncTask {
  List result = null;
  void doInBackground(String key) {
    result = ...
// get from the network
  }
  void onPostExecute() {
    list = result;
    // display the list of titles } }

class DownloadTask extends AsyncTask {
   String content = null;
   void doInBackground(int id) {
      content = ... // get from the network
   }
   void onPostExecute() {
      // display the content } }
```

Fig. 3. A robust program.

on the main thread. When the user input to search for news is triggered, the invocation `doInBackground` of `searchTask` connects to the network, searches for the keyword and fetches the list of resulting news titles. Then, the invocation `onPostExecute` displays the list of titles to the user. `SaveTask` saves the keyword to a database representing the search history in the background. The background tasks `SearchTask.doInBackground` and `SaveTask.doInBackground` might interfere but any interleaving produces the same result, i.e., `searchForNews` is deterministic.

The second event, to show the details of a title, can be triggered once the list of titles are displayed on the screen. It invokes an asynchronous task to download the contents of the news in the background and then displays it. In this case, the tasks are executed in a fixed order and the event is trivially deterministic.

Concerning serializability, the invocation of `SaveTask` in the first event and the second event might interleave (under the concrete semantics). However, assuming that the second event is triggered once the results are displayed, any such interleaving results in the same state as a serial execution of these events.

3 Programs

In order to give a generic definition of robustness, which doesn't depend on any particular asynchronous-programming platform or syntax, we frame our discussion around the abstract notion of programs defined in Sect. 3.1. Two alternative multi-thread and single-thread semantics to programs are given in Sects. 3.2 and 3.3. We consider programs that are *data-deterministic*, in the sense that the evaluation of every (Boolean) expression is uniquely determined by the variable valuation.

3.1 Asynchronous Event-Driven Programs

We define an event handler as a procedure which is invoked in response to a user or a system input. For simplicity, we assume that inputs can arrive in any order. Event handlers may have some asynchronous invocations of other procedures, to be executed later on the same thread or on a background thread.

We fix sets G and L of global and local program states. Local states $\ell \in L$ represent the code and data of an asynchronous procedure or event-handler invocation, including the code and data of all nested synchronous procedure calls. A program is defined as a mapping between pairs of global and local states which gives the semantics of each statement in the code of a procedure (the association between threads, local states, and procedure invocations is defined in Sects. 3.2 and 3.3). To formalize the conflict-based approximation of robustness, this mapping associates with each statement a label called *program action* that records the set of variables read or written and the asynchronous invocations in that statement. An *event set* $E \subset L$ is a set of local states; each $e \in E$ represents the code and data for a single event handler invocation (called event for short).

$x := y \quad y := x \quad \texttt{assume}\, y \quad \texttt{call}\, p(y) \quad \texttt{async}[w]\, p(y) \quad \texttt{return}$

Fig. 4. A canonical program syntax. The metavariables x and y range over global and local variable names, respectively, p ranges over procedure names, and w over the symbols "main" and "any".

Formally, let $X = \{\mathrm{rd}(x), \mathrm{wr}(x) : x \in \ldots\}$ be the set of memory accesses, $W = \{\mathrm{main}, \mathrm{any}\}$ the set of invocation places, and $B = \{\mathrm{invoke}(\ell, w) : \ell \in L, w \in W\} \cup \{\mathrm{return}\} \cup X \cup \{\varepsilon\}$ the set of program actions, where ε represents irrelevant program actions. The $\mathrm{rd}(x)$ and $\mathrm{wr}(x)$ represent read and write accesses to variable x; $\mathrm{invoke}(\ell, w)$ represents an asynchronous invocation whose initial local state is ℓ; the invocation is to be run on a distinguished main thread when $w = \mathrm{main}$, and on an arbitrary thread when $w = \mathrm{any}$. Finally, the return program action represents the return from an asynchronous procedure invocation.

A *program* $P : G \times L \to G \times L \times B$ maps global states $g \in G$ and local states $\ell \in L$ to new states and program actions; each $P(g, \ell)$ represents a single program transition. We assume that when b is an asynchronous invocation or return program action and $P(g, \ell) = \langle g', \_, b\rangle$ then $g = g'$.

Canonical Program Syntax. Supposing that the global states $g \in G$ are maps from program variables x to values $g(x)$, and that local states $\ell \in L$ map program variables y to values $\ell(y)$ and a program counter variable pc to program statements $\ell(\mathrm{pc})$, we give an interpretation to the canonical program syntax listed in Fig. 4. We assume atomicity of the statements at the bytecode level. For simplicity, we omit the interpretation of synchronous procedure calls `call` $p(y)$ which is defined as usual. For instance, writing ℓ^+ to denote $\ell[\mathrm{pc} \mapsto \ell(\mathrm{pc}){+}1]$, then $P(g, \ell)$ is

- $\langle g[x{\mapsto}\ell(y)], \ell^+, \mathrm{wr}(x)\rangle$ when $\ell(\mathrm{pc})$ is a global-variable write x `:=` y,
- $\langle g, \ell^+[y{\mapsto}g(x)], \mathrm{rd}(x)\rangle$ when $\ell(\mathrm{pc})$ is a global-variable read y `:=` x,
- $\langle g, \ell^+, \mathrm{rd}(y)\rangle$ when $\ell(\mathrm{pc})$ is `assume(`y`)` and $\ell(y) \neq 0$,
- $\langle g, \ell, \varepsilon\rangle$ when $\ell(\mathrm{pc})$ is `assume(`y`)` and $\ell(y) = 0$,
- $\langle g, \ell^+, \mathrm{invoke}(\ell', w)\rangle$ when $\ell(\mathrm{pc})$ is an asynchronous invocation `async`$[w]\, p(y)$, where ℓ' maps the parameters of procedure p to the invocation arguments y and pc to the initial statement of p, and
- $\langle g, \ell, \mathrm{return}\rangle$ when $\ell(\mathrm{pc})$ is the `return` statement.

The semantics of other statements, including if-then-else conditionals, while loops, or goto statements, etc. (we assume that Boolean conditions use only local variables), is standard, and yield the empty program action ε.

An event is called *sequential* when its code doesn't contain asynchronous invocations `async`$[w]\, p(y)$. Also, a program P with event set E is called *sequential* when every event $e \in E$ is sequential. Otherwise, P is called *concurrent.*

3.2 Multi-thread Asynchronous Semantics

Our multi-thread semantics maximizes the set of possible program behaviors by allowing events to interleave and interfere with each other. It dispatches the event handlers serially on the main thread but allows the asynchronous procedure invocations to execute on separate threads, not necessarily in invocation order. Configurations of the multi-thread semantics thus maintain sets of running procedure invocations as well as an unordered queue of pending invocations, and invocations are associated with events and threads.

To characterize executions by the event-serializability and event-determinism criteria, we expose the following set A of actions in execution traces:

$$A = \{\text{start}(j), \text{end}(j) : j \in \mathbb{N}\} \cup X \cup \{\text{invoke}(i), \text{begin}(i), \text{return}(i) : i \in \mathbb{N}\}$$

By convention, we denote asynchronous procedure invocation, event, and thread identifiers, respectively, with the symbols i, j, k. The $\text{start}(j)$ and $\text{end}(j)$ actions represent the start and end of event j; the $\text{invoke}(i)$, $\text{begin}(i)$, and $\text{return}(i)$ actions represent an asynchronous procedure invocation (when it is added to the queue of pending invocations), the start of i's execution (when it is removed from the queue), and return of i, respectively. The set X of memory accesses is defined as in the program actions of Sect. 3.1.

A *task* $u = \langle \ell, i, j, k \rangle$ is a local state $\ell \in L$ along with invocation, event and thread identifiers $i, j, k \in \mathbb{N}$, and U denotes the set of tasks. We write $\text{invoc}(u)$, $\text{event}(u)$, and $\text{thread}(u)$ to refer to i, j, and k, respectively. A *configuration* $c = \langle g, t, q \rangle$ is a global state $g \in G$ along with sets $t, q \subseteq U$ of running and waiting tasks such that: (1) invocation identifiers are unique, i.e., $\text{invoc}(u_1) \neq \text{invoc}(u_2)$ for all $u_1 \neq u_2 \in t \cup q$, and (2) threads run one task at a time, i.e., $\text{thread}(u_1) \neq \text{thread}(u_2)$ for all $u_1 \neq u_2 \in t$. The set of configurations is denoted by C_m. We say that a thread k is *idle* in c when $k \notin \{\text{thread}(u) : u \in t\}$, and that an identifier i, j, k is *fresh* when $i, j, k \notin \{\alpha(u) : u \in (t \cup q)\}$ for $\alpha \in \{\text{invoc}, \text{event}, \text{thread}\}$, respectively. A configuration is *idle* when all threads are *idle*.

The transition function $\rightarrow$ in Fig. 5 is determined by a program P and event set E, and maps a configuration $c_1 \in C_m$ and thread identifier $k \in \mathbb{N}$ to another configuration $c_2 \in C_m$ and label $\lambda = \langle i, j, a \rangle$ where i and j are invocation and event identifiers, and $a \in A$ is an action—we write $\text{invoc}(\lambda)$, $\text{event}(\lambda)$, and $\text{act}(\lambda)$ to refer to i, j, and a, respectively. EVENT transitions mark the beginnings of events. We assume that all events are initiated on thread 0, which is also referred to as the *main* thread. Also, for simplicity, we assume that events

EVENT
$$\frac{e \in E \qquad i, j \text{ are fresh}}{g, t, q \xrightarrow{0, \langle \_, j, \text{start}(j) \rangle} g, t, q \cup \{\langle e, i, j, 0 \rangle\}}$$

ASYNC
$$\frac{P(g, \ell_1) = \langle \_, \ell_1', \text{invoke}(\ell_2, w) \rangle \qquad u_2 = \langle \ell_2, i_2, j, k_2 \rangle \\ i_2 \text{ is fresh} \qquad k_2 \text{ is 0 if } w = \text{main or fresh otherwise}}{g, t \cup \{\langle \ell_1, i, j, k \rangle\}, q \xrightarrow{k, \langle i, j, \text{invoke}(i_2) \rangle} g, t \cup \{\langle \ell_1', i, j, k \rangle\}, q}$$

DISPATCH
$$\frac{u = \langle \ell, i, j, k \rangle \qquad k \text{ is idle}}{g, t, q \cup \{u\} \xrightarrow{k, \langle i, j, \text{begin}(i) \rangle} g, t \cup \{u\}, q}$$

RETURN
$$\frac{P(g, \ell) = \langle \_, \_, \text{return} \rangle \qquad j \in \{\text{event}(u) : u \in t \cup q\}}{g, t \cup \{\langle \ell, i, j, k \rangle\}, q \xrightarrow{k, \langle i, j, \text{return}(i) \rangle} g, t, q}$$

END EVENT
$$\frac{P(g, \ell) = \langle \_, \_, \text{return} \rangle \qquad j \notin \{\text{event}(u) : u \in t \cup q\}}{g, t \cup \{\langle \ell, i, j, k \rangle\}, q \xrightarrow{k, \langle i, j, \text{return}(i) \rangle \quad k, \langle i, j, \text{end}(j) \rangle} g, t, q}$$

LOCAL
$$\frac{P(g, \ell) = \langle g', \ell', a \rangle \qquad a \in \{\varepsilon, \text{rd}(x), \text{wr}(x)\}}{g, t \cup \{\langle \ell, i, j, k \rangle\}, q \xrightarrow{k, \langle i, j, a \rangle} g', t \cup \{\langle \ell', i, j, k \rangle\}, q}$$

Fig. 5. The multi-thread transition function $\rightarrow$ for a program P with event set E.

can be initiated arbitrarily at any time. Adding causality constraints between events, e.g., one event can be initiated only when a certain action has been executed, is possible but tedious. ASYNC transitions create pending asynchronous invocations, DISPATCH transitions begin the execution of pending invocations, and RETURN transitions signal their end (the condition in the right ensures that this is not a return from an event). END EVENT transitions mark the end of an event and by an abuse of notation, they map c_1 and k to a configuration c_2 and two labels, return(i) denoting the end of the asynchronous invocation and end(j) denoting the end of the event. All other transitions are LOCAL.

An *execution* of a program P under the multi-thread semantics with event set E to configuration c_n is a configuration sequence $c_0 c_1 \dots c_n$ such that $c_m \xrightarrow{k_m, \lambda_{m+1}} c_{m+1}$ for $0 \leq m < n$. We say that c_n is reachable in P with E under the multi-thread semantics, and we call the sequence $\lambda_1 \dots \lambda_n$ the *trace* of $c_0 c_1 \dots c_n$. The *reachable states* of P with E, denoted $R_m(P, E)$, is the set of global states in reachable idle configurations. The set of traces of P with E under the multi-thread semantics is denoted by $[\![P, E]\!]_m$. We may omit P when it is understood from the context, and write $[\![E]\!]_m$ instead of $[\![P, E]\!]_m$.

The *call tree* of a trace τ is a ranked tree $CallTree_\tau = \langle V, E, O\rangle$ where V are the invocation identifiers in τ, and the set of edges E contains an edge from i_1 to i_2 whenever i_2 is invoked by i_1, i.e., τ contains a label $\langle i_1, \_, \text{invoke}(i_2)\rangle$. The function $O : E \to \mathbb{N}$ labels each edge (i_1, i_2) with an integer n whenever i_2 is the nth invocation made by i_1, i.e., $\langle i_1, \_, \text{invoke}(i_2)\rangle$ is the nth label of the form $\langle i_1, \_, \text{invoke}(\_)\rangle$ occurring in τ (reading τ from left to right).

3.3 Single-Thread Asynchronous Semantics

Conversely to the multi-thread semantics of Sect. 3.2, our single-thread semantics minimizes the set of possible program behaviors by executing all events and asynchronous invocations on the main thread, the asynchronous procedure invocations being executed in a *fixed* order.

We explain the order in which asynchronous invocations are executed using the event handler `searchForNews` in Fig. 3. This event handler is supposed to add the keyword to the search history only after the fetching of the news containing that keyword succeeds. This expectation corresponds to executing the asynchronous procedures according to the DFS traversal of the call tree. In general, this traversal is relevant because it preserves causality constraints which are imprinted in the structure of the code, like in the case of standard synchronous procedure calls. The DFS traversal of the call tree also has a technical advantage as it corresponds with the call stack semantics of synchronous procedure calls. Note however that this semantics is not equivalent to interpreting asynchronous invocations as synchronous, since the caller finishes before the callee starts. In the formalization of this semantics, the DFS traversal is modeled using a stack of FIFO queues for storing the pending invocations.

The formalization of the single-thread semantics reuses the notions of task and label in Sect. 3.2. Let U_0 be the set of tasks $u = \langle \ell, i, j, 0\rangle$ executing on thread 0. We overload the term *configuration* which in this context is a tuple

EVENT

$$\frac{e \in E \qquad i, j \text{ are fresh}}{g, \bot, \varepsilon \xRightarrow{0, \langle \_, j, \mathrm{start}(j) \rangle} g, \bot, \langle e, i, j, 0 \rangle}$$

END EVENT

$$\frac{P(g, \ell) = \langle \_, \_, \mathrm{return} \rangle}{g, \langle \ell, i, j, k \rangle, \varepsilon \xRightarrow{k, \langle i, j, \mathrm{return}(i) \rangle \;\; k, \langle i, j, \mathrm{end}(j) \rangle} g, \bot, \varepsilon}$$

ASYNC

$$\frac{P(g, \ell_1) = \langle \_, \ell_1', \mathrm{invoke}(\ell_2, w) \rangle \qquad u_2 = \langle \ell_2, i_2, j, 0 \rangle \qquad i_2 \text{ is fresh}}{g, \langle \ell_1, i, j, k \rangle, q \cdot f \xRightarrow{0, \langle i, j, \mathrm{invoke}(i_2) \rangle} g, \langle \ell_1', i, j, k \rangle, q \cdot (f \circ i_2)}$$

DISPATCH

$$\frac{u = \langle \ell, i, j, k \rangle \qquad f = u \circ f' \qquad q' \text{ is } \langle\rangle \text{ if } f' = \langle\rangle \text{ or } f' \cdot \langle\rangle \text{, otherwise}}{g, \bot, q \cdot f \xRightarrow{0, \langle i, j, \mathrm{begin}(i) \rangle} g, u, q \cdot q'}$$

RETURN

$$\frac{P(g, \ell) = \langle \_, \_, \mathrm{return} \rangle \qquad j \in \{\mathrm{event}(u) : u \in q\}}{g, \langle \ell, i, j, k \rangle, q \xRightarrow{k, \langle i, j, \mathrm{return}(i) \rangle} g, \bot, \overline{q}}$$

LOCAL

$$\frac{P(g, \ell) = \langle g', \ell', a \rangle \qquad a \in \{\varepsilon, \mathrm{rd}(x), \mathrm{wr}(x)\}}{g, \langle \ell, i, j, k \rangle, q \xRightarrow{k, \langle i, j, a \rangle} g', \langle \ell', i, j, k \rangle, q}$$

Fig. 6. The single-thread transition function $\Rightarrow$ for a program P with events E (ε and $\langle\rangle$ are the empty sequence and tuple, resp.,). Also, f and f' are tuples, and $\overline{q}$ is obtained by popping a queue from q if this queue is empty, or $\overline{q} = q$, otherwise.

$c = \langle g, u, q \rangle$ where $g \in G$, $u \in (U_0 \cup \{\bot\})$ is a possibly-empty task placeholder (at most one task is running at any moment), and $q \in (\mathsf{Tuples}(U_0))^*$ is a sequence of tuples of tasks (a tuple, resp., a sequence, denotes a FIFO queue, resp., a stack). C_s is the set of configurations of the single-thread semantics. We call $c \in C_s$ *idle* if $u = \bot$.

The transition function $\Rightarrow$ in Fig. 6 is essentially a restriction of $\rightarrow$ where all the procedures run on the main thread, an event begins when there are no pending invocations, and the rules ASYNC and DISPATCH use a stack of FIFO queues for storing pending invocations. The effect of pushing/popping a queue to the stack or enqueuing/dequeueing a task to a queue is represented using the concatenation operation $\cdot$, resp.,$\circ$, for sequences, resp., tuples. Every task created by ASYNC is posted to the *main* thread and it is enqueued in the queue on the top of the stack q. DISPATCH dequeues a pending task from the queue f on the top of q, and pushes a new *empty* queue to q (for storing the tasks created during the newly started invocation) if f doesn't become empty. Moreover, the rules RETURN and END EVENT pop the queue on the top of q if it is empty.

An *execution* of a program P under the single-thread semantics with event set E to configuration c_n is a sequence $c_0 c_1 \ldots c_n$ s.t. $c_m \xRightarrow{0, \lambda_{m+1}} c_{m+1}$ for $0 \leq m < n$. We say that c_n is *reachable* in P with E under the single-thread semantics, and we call the sequence $\lambda_1 \ldots \lambda_n$ the *trace* of $c_0 c_1 \ldots c_n$. The *reachable states* of P with E, denoted $R_s(P, E)$, is the set of global states reachable in idle configurations.

The set of traces of P with E under the single-thread semantics is denoted by $[\![P, E]\!]_s$ (P may be omitted when it is understood from the context).

4 Robustness of Asynchronous Programs

Our robustness criterion is defined as the equality of the single-thread and multi-thread semantics of a program, and decomposed into two independently-checkable criteria, event serializability and event determinism.

Given a program P with event set E, each execution under the single-thread semantics can be simulated by an execution under the multi-thread semantics: the latter corresponds to a special scheduling policy that consists in executing all tasks created by an event before starting executing tasks corresponding to another event, and moreover, tasks are executed atomically, in the order given by the DFS traversal of the call tree. This implies that the multi-thread semantics is a relaxation of the single-thread semantics, and therefore, $R_s(P, E) \subseteq R_m(P, E)$. The reverse direction is the most interesting one:

Definition 1 (Robustness). *A program P with events E is* robust against concurrency *(or simply* robust*) when all reachable states in the multi-thread semantics are also reachable in the single-thread semantics: $R_m(P, E) \subseteq R_s(P, E)$.*

Robustness means that for the considered program, the concurrency introduced by the multi-thread semantics does not modify the set of observable states, i.e., $R_m(P, E) = R_s(P, E)$. We introduce in the following two correctness criteria that capture precisely the notion of robustness.

We say an execution with trace $\lambda_1 \cdots \lambda_n$ is *event-serial* when for all $n_1 < n_3$, if $\mathrm{act}(\lambda_{n_1}) = \mathrm{start}(j)$ and $\mathrm{act}(\lambda_{n_3}) = \mathrm{start}(j')$, then there is n_2 such that $n_1 < n_2 < n_3$ and $\mathrm{act}(\lambda_{n_2}) = \mathrm{end}(j)$.

Definition 2 (Event-serializability). *A program P with events E is* event-serializable *if every global state in $R_m(P, E)$ can be reached by an event-serial execution*[1]*.*

Given an event e, an *e-execution starting from global state g_0* is a g_0-initialized execution (according to the multi-thread semantics) with trace $\lambda_1 \cdots \lambda_n$ such that (1) $act(\lambda_1) = \mathrm{start}(j)$, (2) $act(\lambda_n) = \mathrm{end}(j)$, for some j, and (3) for every $m \in \mathbb{N}$ such that $1 < m < n$, $act(\lambda_m)$ is neither a start nor an end action. Intuitively, we consider executions of individual events, from their starting point until the completion of all the tasks they have created. Then, let $R_m(P, g_0, e)$ be the set of global states in final configurations of e-executions starting from g_0. Notice that e-executions from g_0 differ by the scheduling order of the tasks created by e that are running in parallel on different threads.

Definition 3 (Event-determinism). *An event e of a program P is* deterministic *if for every global state g_0, the set $R_m(P, g_0, e)$ is a singleton or empty. A program P with events E is* event-deterministic*, if every $e \in E$ is deterministic.*

Notice that our notion of determinism is defined for events that are running alone, without interference of other events.

Theorem 1. *A program is robust against asynchrony if and only if it is event-serializable and event-deterministic.*

[1] For simplicity, we have ignored the set of events which are executed when comparing global state reached by aribitrary and event-serial executions, resp. Reaching a global state using the same set of events is easy to formalize but tedious.

5 Conflict Robustness

Following an idea introduced in the context of database transactions [27], we define a syntactic, conservative notion of *conflict robustness* that is the conjunction of two properties: *conflict-event serializability* and *conflict-event determinism.*

5.1 Conflict-Event Serializability

Let $\prec \subseteq A \times A$ be a *conflict relation* that relates any two actions a, a' accessing the same variable, i.e., $a, a' \in \{\mathrm{rd}(x), \mathrm{wr}(x)\}$ for some x, one of them being a write. A trace is conflict-event serializable iff the "conflict-event graph" which tracks the conflict relation between concurrent events is acyclic.

Formally, the *conflict-event graph* of a trace τ is the directed graph $EvG_\tau = \langle V, E\rangle$ whose nodes V are the event identifiers of τ, and which contains an edge from j_1 to j_2 when τ contains a pair of labels λ_1 and λ_2 such that λ_1 occurs before λ_2, $act(\lambda_1) \prec act(\lambda_2)$, $event(\lambda_1) = j_1$, and $event(\lambda_2) = j_2$.

Definition 4. *A trace τ is called conflict-event serializable when EvG_τ is acyclic. A program P with event set E is conflict-event serializable iff every trace in $[\![P, E]\!]_m$ is conflict serializable.*

A permutation τ' of a trace τ is *conflict-preserving* when every pair λ_1, λ_2 of labels in τ appear in the same order in τ' whenever $\mathrm{act}(\lambda_1) \prec \mathrm{act}(\lambda_2)$. Note that a conflict-preserving permutation τ' leads to the same global state as the original trace τ. *From now on, whenever we use permutation we mean conflict-preserving permutation.* A trace τ is conflict-event serializable iff it is a conflict-preserving permutation of an event-serial trace.

Theorem 2. *A program P with event set E is event-serializable when it is conflict-event serializable.*

5.2 Conflict Determinism

We define *conflict determinism*, which is also based on the acyclicity of a certain class of "conflict graphs", called *conflict-invocation graphs.* These graphs represent the conflicts between the asynchronous invocations, but also the order in which these invocations would be executed under the single-thread semantics, i.e., the DFS traversal of the call tree. If the conflict-invocation graph of every trace τ of an event e is acyclic, then e is deterministic because every trace τ is a conflict-preserving permutation of the trace t_0 corresponding to the single-thread semantics, and thus leads to the same global state as t_0.

Given a trace τ, let $<_{dfs}$ be the total order between the invocation identifiers in τ defined by the DFS traversal of $CallTree_\tau$. The *conflict-invocation graph* of a trace τ is the directed graph $InvG(\tau) = \langle V, E\rangle$ whose nodes V are the asynchronous invocation identifiers in τ, and which contains an edge from i_1 to i_2 when $i_1 <_{dfs} i_2$, or τ contains a pair of labels λ_1 and λ_2 of i_1 and i_2, resp., such that $act(\lambda_1) \prec act(\lambda_2)$ and λ_1 occurs before λ_2.

Definition 5. *A trace τ is DFS-serial iff $InvG(\tau)$ is acyclic. An event e is* conflict-deterministic *iff every trace in $[\![e]\!]_m$ is DFS-serial.*

A trace τ is called *invocation-serial* iff for every three labels $\lambda_1, \lambda_2, \lambda_3$ occurring in τ in this order, if $invoc(\lambda_1) = invoc(\lambda_3)$, then $invoc(\lambda_1) = invoc(\lambda_2)$. For an event e, a DFS-serial trace τ in $[\![e]\!]_m$ is a permutation of an invocation-serial trace $\tau_0 \in [\![e]\!]_m$ where $\text{invoc}(\lambda_1) <_{dfs} \text{invoc}(\lambda_2)$ for every two labels λ_1 and λ_2 occurring in this order in τ_0.

Theorem 3. *An event is deterministic when it is conflict-deterministic.*

6 Checking Conflict Determinism

We reduce the problem of checking conflict determinism of an event to a reachability problem in a *sequential* program. We present the reduction in two steps. First, conflict determinism of an event interpreted under the multi-thread semantics, whose asynchronous invocations run concurrently, is reduced to a reachability problem in a program running on the single-thread semantics, where asynchronous invocations are executed serially (Sects. 6.1 and 6.2). The latter is then reduced to a reachability problem in a sequential program (Sect. 6.3).

This reduction uses the fact that a certain class of conflict determinism violations can be simulated by a sequential program up to conflict-preserving permutations of actions (note that any conflict-preserving permutation of a violation is also a violation). This class of violations called *borderline violations* are minimal in the sense that removing the last action leads to a correct trace. Besides the simulation, we show that fixed-size additional memory is required to witness the conflicts inducing a cycle in the conflict invocation graph.

Definition 6 (Borderline Conflict Determinism Violation). *A trace τ is a* borderline violation *to conflict determinism if it is not DFS-serial but every strict prefix of τ is DFS-serial.*

For instance, the trace τ_1 given in Fig. 7(a) contains a borderline violation. This trace is generated by an event e that invokes two procedures p and q in this order, each procedure on a different thread. The only conflict between memory accesses is that between the $wr(x)$ actions in q and resp., p. The conflict-invocation graph of τ_1 contains a cycle between the invocations of p and q: the edge from the invocation of p to that of q is implied by the fact that p is invoked before q within the same procedure (we have "$p <_{dfs} q$"), and the edge in the other direction exists because q writes to the variable x before p does. The trace τ_1 until after the second $wr(x)$ is a borderline violation since its maximal strict prefix (without the second $wr(x)$) is DFS-serial. The last label of a borderline violation τ, in this example $wr(x)$, is called the *pivot* of τ. The label of τ which precedes and conflicts with its pivot and which induces the cycle in its conflict-invocation graph is called the *root* of τ. Formally, if i_1 is the invocation containing the pivot of τ, the root of τ is an action conflicting with the pivot and which is included in an invocation i_2 such that $i_1 <_{dfs} i_2$. For the trace in Fig. 7(a), the root is the action $wr(x)$ in the invocation of q.

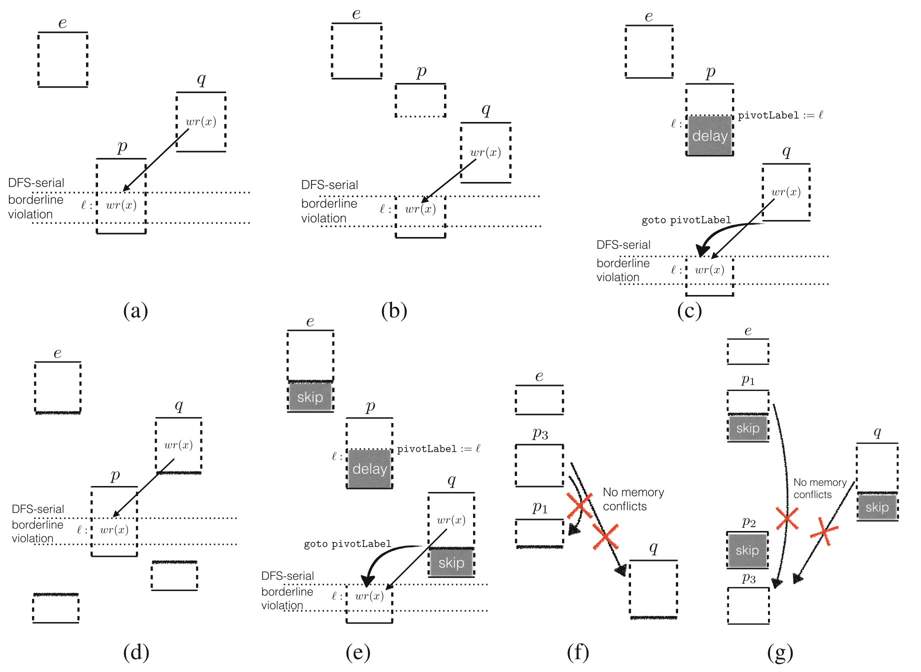

Fig. 7. Simulating borderline conflict determinism violations on the single-thread semantics. The event e makes two fresh thread asynchronous invocations to p and q in this order. Boxes represent sequences of trace labels ordered from top to bottom. Actions of the same thread are aligned vertically. The arrows represents transition label conflicts. For readability, we omit the event and task identifiers in the trace labels and keep only the memory accesses. The grey blocks labeled by delay, resp., skip, denote sequences of actions that are delayed, resp., skipped.

6.1 Simulating Borderline Violations

We define a code-to-code translation from an event e to an event $\mathsf{detStr}^-(e)$ which simulates[2] permutations of every DFS-serial or borderline violation trace in $[\![e]\!]_m$. The event $\mathsf{detStr}^-(e)$ uses additional non-deterministically enabled statements to simulate the particular interleavings present in those traces. The instrumentation required to witness violations is introduced in Sect. 6.2.

Overview. We give an informal description of the translation using as examples the traces pictured in Fig. 7.

Delaying the Pivot. We first explain the simulation of the invocation that contains the pivot, which may interfere with invocations that are supposed to be

[2] We refer to the standard notion of (stuttering) simulation where (sequences of) transitions in $\mathsf{detStr}^-(e)$ are mapped to transitions of e.

executed *later* under the single-thread semantics. For the borderline violation in Fig. 7(a), the invocation of p that contains the pivot $wr(x)$ destroys the value written to x by q, an invocation which is executed after p under the single-thread semantics.

The maximal strict prefix (ending before the second $wr(x)$) is DFS-serial and can be reordered to a trace where the order between transition labels is consistent with the invocation order (i.e., e before p and before q). Figure 7(b) pictures such a reordering, denoted by τ_1'. Our goal is to show that the trace τ_1' can be simulated by an execution under the single-thread semantics of a slightly modified version of e. First note that τ_1' is not admitted by the single-thread semantics of e because the invocation of p is only *partially* included in this prefix. And the single-thread semantics executes every task until completion. However, it is possible to "delay" the execution of the pivot $wr(x)$ in p until q finishes, even under the single-thread semantics, by adding a suitable set of auxiliary variables to e. This mechanism is pictured in Fig. 7(c). Every statement in the procedure p is guarded by (the negation of) an auxiliary Boolean flag `skip` which can be *non-deterministically* flipped to `true` in order to skip over statements. Moreover, an auxiliary global variable `pivotLabel` will record the next control flow label ℓ when this flag is set to `true`. Then, extending the invocation of q with `goto pivotLabel` allows to resume the invocation of p and execute the pivot. To simulate every borderline violation, the `goto` statement is non-deterministically enabled in every invocation.

Incomplete Invocations. While the violation in Fig. 7(a) includes only one incomplete invocation (the one containing the pivot) this is not always the case. A borderline violation may contain unboundedly-many other incomplete invocations. For instance, the violation in Fig. 7(d) includes incomplete invocations of e and q (they finish after the pivot). Should the simulation of this borderline violation execute e and q entirely, the pivot may never be enabled. The correct simulation, pictured in Fig. 7(e), will make use of the same mechanism based on the Boolean flag `skip` in order to skip over statements in e and q. In general, an invocation can be skipped in its entirety. This simulation also shows that the `goto` statement can be executed after an incomplete invocation.

Main Thread Invocations. The last issue concerns the main thread which has the particularity of being able to execute more than one invocation (all the other threads execute a single invocation). It executes invocations serially and only the last one may be incomplete. For instance, consider the DFS-serial trace τ_3 pictured in Fig. 7(f). This is the trace of an event e that invokes p_1, q, p_2, and p_3, in this order, and except q all the tasks are assigned to the main thread. Since p_1 is invoked before p_3, a DFS-serial permutation τ_3' of τ_3 contains the *incomplete* invocation of p_1 before the *complete* invocation of p_3, as shown in Fig. 7(g). None of the semantics we defined allows such traces. The problem is that both invocations are executed by the main thread which has to complete a task before executing another one. Our simulation will however admit such traces but it will verify that they are conflict-preserving permutations of valid traces. This verification procedure (included in the definition of $\mathsf{detStr}^{-}(e)$) checks that the conflict invocation graph doesn't contain a path of *memory conflicts*, i.e., conflicts induced by read and write accesses, from the incomplete invocation on the main thread to any future complete invocation on the same thread. Let us consider again the trace τ_3 in Fig. 7(f). Since τ_3 is DFS-serial, its conflict invocation graph doesn't contain paths of memory conflicts from p_3 to any other invocation ordered before p_3 in the DFS traversal of the call tree. This includes the incomplete invocation p_1 and q. For the permutation τ_3', this implies that its conflict invocation graph contains no paths of memory conflicts from p_1 to p_3. When a trace satisfies this condition, i.e., an incomplete invocation on the main thread doesn't conflict with a future complete invocation on the same thread, all the complete invocations on the main thread can be reordered before the incomplete one (preserving the order between conflicting trace labels) and this results in a valid trace (under the multi-thread semantics). The simulation of τ_3' on the single-thread semantics, pictured in Fig. 7(g), enables this verification procedure during the invocation of p_1 because it is executed on the main thread and it skips over statements. It is also possible that other invocations on the main thread, e.g., p_2, are skipped *entirely.*

Notations. We introduce several notations used in the definition of $\texttt{detStr}^-(e)$. This event is obtained by rewriting every statement s of a procedure transitively invoked by e to a code fragment $s_1; \texttt{if}(c)\ \texttt{then}\ s; s_2$ where s_1 and s_2 are statements and c is a Boolean expression. We use before(s), guard(s), and after(s) to refer to s_1, c, and s_2, respectively. For every statement s, $\ell(s)$ denotes the control flow label of s, that can be used for instance in goto statements. Also, $\mathit{rdSet}(s)$, resp., $\mathit{wrSet}(s)$, is the set of global variables read, resp., written, by s. We have $\mathit{wrSet}(s) = \{x\}$ and $\mathit{rdSet}(s) = \emptyset$ when s is $x := y$, and $\mathit{wrSet}(s) = \emptyset$ and $\mathit{rdSet}(s) = \{x\}$ when s is $y := x$. Otherwise, $\mathit{rdSet}(s) = \mathit{wrSet}(s) = \emptyset$.

We assume that every procedure p is augmented with two local variables `rdSetProc` and `wrSetProc` tracking the global variables read and written by p, respectively ($\mathit{rdSet}(s)$ and $\mathit{wrSet}(s)$ are added to `rdSetProc` and `wrSetProc`, respectively, after every statement s that gets executed).

The instrumentation uses the non-deterministic choice denoted by $*$ (formally, $*$ is a distinguished Boolean variable that evaluates non-deterministically to `true` or `false`). To refer to the different non-deterministic choices in the instrumentation, we may index them with natural numbers.

To reduce clutter in the instrumentation, we use $[\,s\,]_{\texttt{ev}}(\texttt{b})$ to denote a statement s that is executed at most once during the execution of the event and the Boolean variable `b` is set to `true` when s gets executed.

For an event e, let $\mathcal{P}(e)$ be the set of the procedures possibly invoked by e, which is defined inductively by: (1) $e \in \mathcal{P}(e)$ and (2) for every $p \in \mathcal{P}(e)$, if $\texttt{async}[w]\ q(y)$ occurs syntactically in the code of e, then $q \in \mathcal{P}(e)$. Also, let $\mathcal{P}_0(e)$ be the subset of $\mathcal{P}(e)$ consisting of procedures posted to the main thread, i.e., in the previous inductive definition, we take $w = \texttt{main}$. W.l.o.g. we assume that the procedures in $\mathcal{P}_0(e)$ are distinct from the procedures q contained in asynchronous invocations "$\texttt{async}[\text{any}]\ q(\ldots)$" executed on other threads.

All the Boolean variables added by the instrumentation are initially `false`.

Defining the Instrumentation

Dealing with Fresh Thread Invocations. To simulate incomplete invocations executed by threads other than the main thread, every procedure in $\mathcal{P}(e) \setminus \mathcal{P}_0(e)$ is augmented with a Boolean flag `skip` that is non-deterministically set to `true`. Once `skip` is set to `true`, the rest of statements are skipped and the first skipped statement *may* be chosen as the pivot and its label stored in `pivotLabel`. The pivot may get executed non-deterministically at a later time.

The program instrumentation to simulate borderline violations is given in Fig. 8a. For every statement s of procedure $p \in \mathcal{P}(e) \setminus \mathcal{P}_0(e)$, guard$(s)$ and before(s) are defined respectively at lines 1 and 4 where `skip` is a local variable and `pivotLabel` is a global variable.

Dealing with Main Thread Invocations. For procedures in $\mathcal{P}_0(e)$, the instrumentation ensures that at most one invocation of such a procedure is incomplete, and also, that the invocation graph contains no path of memory conflicts from such an incomplete invocation to any future complete invocation of a procedure in $\mathcal{P}_0(e)$. Such paths of memory conflicts may cross invocations of procedures which are not in $\mathcal{P}_0(e)$, therefore the instrumentation of the latter must also be modified.

To simulate an incomplete invocation on the main thread, for every statement s of a procedure $p \in \mathcal{P}_0(e)$, before(s) is defined as in line 15 in Fig. 8a where `skip` is a Boolean local variable. As for invocations executed on other threads, the first skipped statement may be chosen as the pivot. To be able to track

```
1   // guard($s$) for $p \in \mathcal{P}(e) \setminus \mathcal{P}_0(e)$:
2   !skip
3
4   // before($s$) for $p \in \mathcal{P}(e) \setminus \mathcal{P}_0(e)$:
5   if ( !skip & $*_1$ ) then
6     skip := true
7     if ( $*_2$ ) then
8       [pivotLabel := $\ell(s)$]$_{ev}$(pivotSet)
9   if ( skip & pivotSet & $*_3$) then
10    [goto pivotLabel]$_{ev}$(gotoDone)
11
12  // guard($s$) for $p \in \mathcal{P}_0(e)$:
13  ! skipProc & ! skip
14
15  // before($s$) for $p \in \mathcal{P}(e)$:
16  if ( $*_4$ ) then
17    [skip := true]$_{ev}$(skipMainSet)
18    rdSetGlobal := rdSetProc
19    wrSetGlobal := wrSetProc
20    if ( $*_5$ ) then
21      [pivotLabel := $\ell(s)$]$_{ev}$(pivotSet)

22  // at the beginning of each $p \in \mathcal{P}(e)$:
23  if ( $*_6$ ) then skipProc := true
24  validMain := false
25
26  // after($s$) for $p \in \mathcal{P}(e)$, after($s$):
27  if ( skipMainSet & ( rdSetGlobal $\cap$ wrSetProc $\neq \emptyset$
28     | wrSetGlobal $\cap$ rdSetProc $\neq \emptyset$
29     | wrSetGlobal $\cap$ wrSetProc $\neq \emptyset$
30     | conflictDetected) ) then
31     rdSetGlobal := rdSetGlobal $\cup$ rdSetProc
32     wrSetGlobal := wrSetGlobal $\cup$ wrSetProc
33     conflictDetected := true
34
35  // at the end of each $p \in \mathcal{P}(e)$:
36  if ( skipMainSet & ! skipProc & ! skip ) then
37    assume ! conflictDetected
38    validMain := true
39  if ( pivotSet & $*_7$ ) then
40    [goto pivotLabel]$_{ev}$(gotoDone)
```

(a) Simulating Borderline Violations

```
41  // added to before($s$):
42  if (! skip & pivotSet & $*$) then
43    [rootLabel := $\ell(s)$]$_{ev}$(rootSet)

44  // added to after($s$):
45  if (conflict(pivotLabel, rootLabel) & pivotLabel == $\ell(s)$
46    & gotoDone & rootSet ) then error := true;
```

(b) Witnessing Borderline Violations

Fig. 8. Instrumentation for checking conflict-determinism.

paths of memory conflicts, the variables read and written during the incomplete invocation are stored in the global variables `rdSetGlobal` and `wrSetGlobal`, respectively. For invocations of procedures $p \in \mathcal{P}_0(e)$, `skip` can be set to `true` at most once during the execution of the event.

Other tasks posted to the main thread can be skipped entirely or executed completely, by setting a local flag `skipProc`. When they are executed completely, a global Boolean flag `validMain` is used to witness that they are not the destination of a path of memory conflicts as explained above. At the beginning of each procedure, `validMain` is reset to `false` as shown at line 22. Then, guard(s) of every statement s of a procedure $p \in \mathcal{P}_0(e)$ checks for `skipProc` as in line 13.

Once an incomplete invocation on the main thread is present, i.e., `skipMainSet` is `true`, the procedure for checking the absence of paths of memory conflicts is enabled. For every statement s of every procedure $p \in \mathcal{P}(e)$, after(s) is set as in line 26 where `conflictDetected` is a Boolean local variable. This conditional checks whether the current procedure conflicts with the incomplete invocation or transitively, with all the other invocations that conflict with the latter. If this is the case, then its set of memory accesses is continuously added to the global sets `rdSetGlobal` and `wrSetGlobal` of memory accesses.

When a main thread invocation finishes, if it has been executed completely and if it follows an incomplete main thread invocation, the instrumentation checks for absence of paths of memory conflicts and may non-deterministically execute the pivot. The code at line 35 is added at the end of every $p \in \mathcal{P}_0(e)$.

Relationship Between e and $\mathsf{detStr}^{-}(e)$*.* The following result expresses the relationship between the original event e and $\mathsf{detStr}^{-}(e)$. It shows that the single-thread semantics of $\mathsf{detStr}^{-}(e)$ simulates permutations of all the DFS-serial traces and borderline violations of e under the multi-thread semantics (modulo a thread id renaming). Moreover, every trace of $\mathsf{detStr}^{-}(e)$ under the single-thread semantics where the last value of `validMain` is `true`, this set of traces being denoted by $[\![\mathsf{detStr}^{-}(e)]\!]_s^{\mathtt{validMain}}$, corresponds to a trace of e under the multi-thread semantics (modulo the instrumentation added in $\mathsf{detStr}^{-}(e)$ and a thread id renaming). For a trace τ of $\mathsf{detStr}^{-}(e)$, $\overline{\tau}$ is the trace obtained from τ by erasing all transition labels corresponding to statements added by the instrumentation. For readability, we ignore the issue of renaming thread ids.

Theorem 4. *For every trace τ_1 in $[\![e]\!]_m$, if τ_1 is DFS-serial or a borderline conflict determinism violation, then there exists a trace τ_2 in $[\![\mathsf{detStr}^{-}(e)]\!]_s$ such that $\tau_1' = \overline{\tau_2}$ is a conflict-preserving permutation of τ_1. Moreover, for every trace τ_1 in $[\![\mathsf{detStr}^{-}(e)]\!]_s^{\mathtt{validMain}}$ there exists a trace τ_2 in $[\![e]\!]_m$ such that $\tau_2 = \overline{\tau_1}$.*

6.2 Witnessing Borderline Violations

The instrumentation used to verify that a trace is indeed a borderline violation consists in guessing a candidate for the root and then, when the pivot gets executed, checking whether it conflicts with the chosen candidate. For instance, if we consider the single-thread semantics simulation in Fig. 7(c), the action

$wr(x)$ in q is guessed as the root and its label is stored in an auxiliary variable `rootLabel`. This label is used to check that the root candidate conflicts with the pivot when the latter is executed. The root must be chosen after the pivot in order to guarantee that this leads to a cycle in the conflict invocation graph (i.e., the DFS traversal of the call tree orders the invocation containing the pivot before the one containing the root).

We define a new event $\mathsf{detStr}(e)$ that sets an `error` flag to `true` whenever the current trace is not DFS serial and the root and pivot candidates are valid. This event is obtained from $\mathsf{detStr}^-(e)$ by adding two global variables `error` and `rootLabel`, and:

– Concatenating the code at line 41 in Fig. 8b to before(s). This allows to non-deterministically choose s to be the root of the violation. In order to avoid choosing the pivot after the root, we must also replace $*_2$ and $*_5$ in $\mathsf{detStr}^-(e)$ with ! `rootSet` & $*_2$ and ! `rootSet` & $*_5$, respectively.
– Concatenating the code at line 44 in Fig. 8b to after(s) where

$$\begin{aligned} \mathit{conflict}(\texttt{pivotLabel}, \texttt{rootLabel}) \ ::= \ & \mathit{rdSet}(\ell^{-1}(\texttt{pivotLabel})) \cap \mathit{wrSet}(\ell^{-1}(\texttt{rootLabel})) \neq \emptyset \\ & \| \mathit{rdSet}(\ell^{-1}(\texttt{rootLabel})) \cap \mathit{wrSet}(\ell^{-1}(\texttt{pivotLabel})) \neq \emptyset \\ & \| \mathit{wrSet}(\ell^{-1}(\texttt{rootLabel})) \cap \mathit{wrSet}(\ell^{-1}(\texttt{pivotLabel})) \neq \emptyset \end{aligned}$$

This allows to validate that the root does indeed conflict with the pivot, once the latter gets executed. If the conflict is validated, then `error` is set to `true`.

Since the added instrumentation only reads variables of $\mathsf{detStr}^-(e)$, the new event $\mathsf{detStr}(e)$ still satisfies the claim in Theorem 4.

Theorem 5. *An event e (under the multi-thread semantics) satisfies conflict determinism iff the program* $\mathsf{detStr}(e)$ *under the single-thread semantics does not reach a state where* `error` = `true`.

For complexity, $\mathsf{detStr}(e)$ can be constructed in linear time and its number of variables increases linearly in the number of variables and procedures of e.

6.3 Reduction to the Procedural Semantics

As a continuation to Theorem 5, we define a code-to-code translation from an event e to a *sequential* event $\mathsf{seq}(e)$ such that $\mathsf{seq}(e)$ admits exactly the set of traces of e under the single-thread semantics[3].

Single-Thread Semantics vs Procedural Semantics. Essentially, $\mathsf{seq}(e)$ is obtained from e by rewriting asynchronous procedure invocations to regular procedure

[3] Modulo the omission of the labels invoke(i), begin(i), return(i) related to asynchronous invocations.

calls. However, this rewriting can't be applied directly because of the following issue. Consider a procedure p invoking another procedure q. If the invocation of q is asynchronous, the single-thread semantics executes p completely before starting q. Under the procedural semantics, when q is invoked using a regular procedure call, the execution of p is blocked when q is invoked and resumed when q is completed. For instance, consider the event:

```
procedure e1(){y:=1;async[main] p();y:=2;} procedure p(){y:=3;}
```

Executing e_1 on the single-thread semantics, we get the sequence of assignments `y := 1`, `y := 2`, `y := 3`. Rewriting `async[main]` $p()$ to a regular procedure call `call` $p()$, we get an event that executes `y := 1`, `y := 3`, `y := 2` in this order.

This issue doesn't exist if all the asynchronous invocations occur at the end of the procedures. For instance consider the following event e_2:

```
procedure e2(){x:=1;async[main] p();}  procedure q() {x:=3;}
procedure p(){x:=2;async[main] q();}
```

Rewriting every `async[main]` _ to a procedure call `call` _, we get an event that executes the assignments on x in exactly the same order as e_2 under the single-thread semantics. This holds because the single-thread semantics executes the asynchronous invocations according to the DFS traversal of the call tree, which corresponds to the "stack" semantics of procedure calls.

Therefore, the event $\mathsf{seq}(e)$ is obtained in two steps. A first translation is used to move all asynchronous invocations at the end of the procedures. This results in an event having exactly the same single-thread semantics as the original one. Then, we replace every asynchronous invocation with a procedure call.

Defining $\mathsf{seq}(e)$*.* The event e is extended with auxiliary data structures that store the names and the inputs of the asynchronous invocations. Using these data structures, all the invocations are delayed till the end of the encompassing procedure. Thus,

– each procedure p is extended with an auxiliary local variable `invocList` which stores a list of procedure names and inputs,
– when an asynchronous procedure q is invoked in p with inputs y, the procedure name q together with its parameters y is appended to the local variable `invocList` of p without invoking q,
– before returning from a procedure p, all the procedures stored in `invocList` are invoked in the order they are recorded.

For the event e_1, this boils down to simply moving the invocation in e_1 at the end (i.e., after `y := 1`). It is easy to see that the obtained event has the same single-thread semantics as the original event.

Let $\mathsf{seq}(e)$ be the event obtained from e by applying the transformation above and then, replacing every asynchronous invocation $\texttt{async}[w]\ p(y)$ with $\texttt{call}$ $p(y)$.

For an event e, we overload the equality relation between traces $\tau_1 \in [\![e]\!]_s$ and $\tau_2 \in [\![\mathsf{seq}(e)]\!]_s$ as follows: $\tau_1 = \tau_2$ iff removing the labels invoke(i), begin(i), return(i) with $i \in \mathbb{N}$ from τ_1, and the transition labels corresponding to statements added by the instrumentation from τ_2, we get the same trace.

A sequential program Seq has the same set of traces under the multi-thread and the single-thread semantics, so its set of traces is denoted $[\![\mathsf{Seq}]\!]$.

Theorem 6. *For any event e, $[\![e]\!]_s = [\![\mathsf{seq}(e)]\!]$.*

For an event e, let $\mathsf{detSeq}(e) = \mathsf{seq}(\mathsf{detStr}(e))$. By Theorem 6, $\mathsf{detSeq}(e)$ still satisfies the claim in Theorem 4. The following is a direct consequence of Theorems 5 and 6.

Corollary 1. *An event e (under the multi-thread semantics) satisfies conflict determinism iff the sequential event* $\mathsf{detSeq}(e)$ *does not reach a state where $\mathit{error} = \mathit{true}$.*

Concerning complexity, let e be an event where each procedure invokes at most k other procedures, for some fixed k. Then, the time complexity of constructing $\mathsf{detSeq}(e)$ and its number of variables are quadratic in the number of variables and procedures of e and k.

7 Checking Conflict Robustness

Building on the reduction of conflict determinism to reachability in sequential programs, we show that a similar reduction can be obtained for conflict robustness. This reduction is based on two facts: (1) incomplete executions of conflict-deterministic events can be simulated by a sequential program, which has been proved in Sect. 6, and (2) conflict serializability for a set of conflict deterministic events can be again reduced to reachability in sequential programs. To prove the latter we use the concept of borderline violation, this time for conflict serializability. We show that interleavings corresponding to such violations can be simulated by a sequential program. This program behaves like a "most-general client" of the event-based program in the sense that it executes an arbitrary set of events, in an arbitrary order, but serially without interference from others. We show that the memory required to track the conflicts which induce a cycle in the conflict graph is of *bounded* size, although the conflict graph cycles are of unbounded size in general.

Definition 7 (Borderline Conflict Serializability Violation). *A trace t is a* borderline violation *to conflict serializability if it is not conflict serializable but every strict prefix of τ is conflict serializable.*

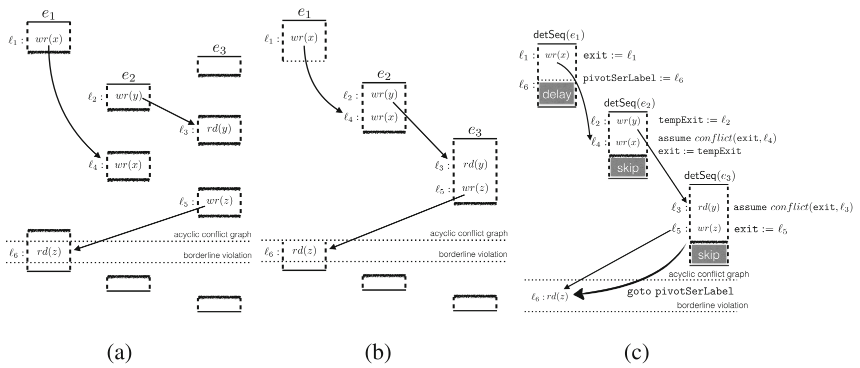

Fig. 9. Simulating borderline conflict serializability violations with a sequential program. Boxes represent sequences of trace labels ordered from top to bottom. Actions of the same event are aligned vertically. The arrows represent all the conflicts in the trace. The grey blocks labeled by delay, resp., skip, denote sequences of actions that are delayed, resp., skipped.

The trace τ_1 in Fig. 9(a) contains a borderline violation. Its conflict-event graph contains a cycle between the three events e_1, e_2, and e_3. The prefix of τ_1 ending just before $rd(z)$ satisfies conflict serializability. The last label of a borderline violation τ is called the *pivot* of τ (in this example $rd(z)$) and the event that contains the pivot is called the *delayed event* of τ (in this example e_1).

Simulating Borderline Violations. For a set of conflict-deterministic events E, we define a code-to-code translation to a set of sequential events that simulates every conflict-serializable trace and every borderline serializability violation of E under the multi-thread semantics.

As for conflict determinism, the maximal *strict* prefix of a borderline violation can be reordered to a trace where events are executed serially, but *possibly not until completion* (because it satisfies conflict serializability). Such a reordering for the trace τ_1 is given in Fig. 9(b). This reordering can be simulated by a sequential program that executes the conflict determinism instrumentations $\mathsf{detSeq}(e_i)$ with $i \in [1, 3]$ instead of the original events, as shown in Fig. 9(c). The sequential program chooses non-deterministically the delayed event, in this case e_1, and the pivot, and stores the latter in an auxiliary variable `pivotSerLabel` when leaving the delayed event. While executing other possibly incomplete events using the skipping mechanism introduced for conflict determinism, it may non-deterministically choose to execute goto `pivotSerLabel`, in this case after e_3.

Witnessing Borderline Violations. To establish that a trace is indeed a borderline violation, the instrumentation guesses for each event a statement called *exit point* which conflicts with an action of a future event and a statement called *entry point* which conflicts with the currently recorded exit point of a previous event. The conflict is validated each time an entry point is chosen. This instrumentation is demonstrated in Fig. 9(c). For instance, while simulating e_1, $wr(x)$ is guessed as the exit point and its label is recorded in the auxiliary `exit` variable. During the simulation of e_2, $wr(x)$ is guessed as the entry point and the conflict is validated. As the simulation of e_2 shows, the exit point may occur before the entry point. In this case, the instrumentation uses an additional variable `tempExit` to store the exit point of the current event until the conflict with a previous event is validated. Once the conflict is confirmed the value of `tempExit` is copied to `exit`. Since the conflicts must form a path in the conflict event graph, there is no need to recall more than one exit point at a time.

The instrumentation added for checking conflict robustness is similar to the one used for conflict determinism. Let $\mathsf{robSeq}(e)$ denote the sequential event obtained from $\mathsf{detSeq}(e)$ by adding this instrumentation. For an event set E, let $\mathsf{robSeq}(E) = \{\mathsf{robSeq}(e) : e \in E\}$.

Then, let $\mathsf{robSeq}(E)$ be the set of events $\mathsf{robSeq}(e)$ with $e \in E$.

Theorem 7. *A program P with events E satisfies conflict robustness iff $\mathsf{robSeq}(E)$ doesn't reach a state where `error` $=$ `true`.*

For complexity, $\mathsf{robSeq}(E)$ can be constructed in linear time and the number of additional variables is linear in the number of procedures in $\mathsf{detSeq}(E)$. The complexity of checking conflict robustness is given by the following theorem.

Theorem 8. *Checking conflict robustness of a program P with events E, a fixed number of variables which are all Boolean, and a fixed number of procedures, each procedure containing a fixed number of asynchronous invocations, is polynomial time decidable.*

8 Experimental Evaluation

The goal of our experimental work [5] is to show that (i) event-serializability and event-determinism violations correspond to actual bugs, and (ii) detecting these violations using the reduction to reachability in sequential programs is feasible.

We use the Soot framework [7] to implement the instrumentation required for robustness checking. The reachability of the error state in the instrumented sequential program is verified using Java Path Finder (JPF) [4].

We applied the conflict-robustness checking algorithm to a set of Android apps from the FDroid [3] repository. The application code for reflection, dependency to external libraries (e.g., for http connection, analytics tracker, maps), and the code which only effects the display (e.g., displaying web pages, animation, custom graphics) is eliminated. The remaining code factors out the variables that does not effect the concurrent behavior of the program and keeps the program logic.

We define an event as a procedure which is invoked by the Android app in order to initialize an activity, in response to an user input (e.g., clicking on a button, writing text, navigating back) or a system input (e.g., location change, network disconnect). Our tool receives as input a driver class which initiates the application and invokes a set of events. The tool checks conflict-robustness for the set of executions defined by the driver class. In our experiments, we take into consideration causality constraints between events, e.g., the event handler of a UI component can not be invoked if it is not visible on the screen.

8.1 Event-Determinism Experiments

Table 1. Experimental data for conflict determinism. The last column lists whether the event is found conflict deterministic.

Application	Event handler	#inst	#c	#m	#r/w	#(*)	t(m:s)	Det?
aarddict	Create activity	1307780	177	3016	90	1428	0:01	Y
	Lookup word	77203	222	3604	60	103	<1 s	Y
	Scan sd	21334	167	2941	15	21	<1 s	Y
apphangar	Select item	58908	222	3560	48	70	<1 s	Y
	Update icon pack	13308927	264	4004	95	28833	00:33	N
bookworm	Generate cover	34528	194	2928	30	41	<1 s	Y
	Retrieve cover	36789	213	3440	31	41	<1 s	Y
	Save edits	63017	189	3015	108	158	<1 s	Y
	Search book	53250	185	3012	50	69	<1 s	Y
grtgtfs	Fav stops	53995	162	2885	142	113	<1 s	Y
	Process bustimes	65945	159	2749	105	168	<1 s	Y
	Search route	55077	167	2968	34	67	<1 s	Y
	Search stop	56742	168	2968	52	75	0:01	Y
irccloud	Save prefs	103344	293	3478	18	15	<1 s	Y
	Save settings	102868	293	3478	17	13	<1 s	Y
	Select buffer	136224103	379	4330	761	260605	8:04	Y
	Send message	162682	356	4140	171	77	<1 s	Y
vlille	Load stations	971665	404	5808	236	131	0:01	Y
	Load favorites	9583	141	2400	37	0	<1 s	Y
	Update stations	975974	416	5905	265	131	0:01	Y

Table 1 lists the experimental data related to conflict-deterministic checking. Related to the size of the event handlers, we list the number of analyzed instructions (#inst), loaded classes (#c) and methods (#m). The analysis time is affected by the number of resolved non-deterministic data choices (#(*)), the number of asynchronous invocations, whether the instrumented read/write accesses are made in these invocations, and the execution time of the analyzed program.

We have applied our algorithm to various event handlers and all but one are found to be deterministic. A determinism violation is found in `iconPackUpdated` benchmark as explained in Sect. 2. The pivot of the violation is a write access to the `mAdapter` variable by a procedure running in the background, and the root is a read on the same variable made by a procedure running on the main thread.

8.2 Event-Serializability Experiments

Table 2 shows experimental data for conflict-serializability checking.

True Bugs. Four of the benchmarks had traces with conflict serializability violations which we concluded were true bugs (and true event-serializability violations) after examining the code and the consequences of these violations.

Table 2. Experimental data for conflict serializability. The last two columns say whether the example is serializable and whether a violation is not spurious.

Application	Seq.	#inst	#c	#m	#r/w	# (*)	t(m:s)	Ser?	Bug?
aarddict	1	1084371993	224	3620	154	1764359	23:12	N	Y
	2	101776570	169	2957	100	195370	1:42	N	Y
bookworm	1	22701600	183	2801	202	77614	0:42	Y	-
	2	19179949	183	2801	201	61896	0:33	Y	-
	3	1094300968	189	3016	286	3494089	33:51	Y	-
	4	3547795	188	3029	131	15961	0:08	N	Y
grtgtfs	1	74082801	168	2969	123	279857	2:04	Y	-
	2	-	-	-	149	-	>1 h	-	-
	3	1130239	139	2692	77	4712	0:02	Y	-
	4	60736622	170	2984	161	163236	1:21	N	N
irccloud	1	33713083	293	3479	141	147000	2:55	Y	-
	2	1761539	293	3479	140	7851	0:10	Y	-
	3	171715464	294	3485	147	534338	08:51	Y	-
	4	-	-	-	-	2110	>1 h	-	-
	5	-	-	-	-	902	>1 h	-	-
	6	54556857	358	4165	849	208076	5:28	N	Y
	7	11104756	357	4154	833	39599	0:59	N	Y
vlille	1	48935337	406	5824	286	143461	3:05	N	Y
	2	394535226	406	5824	292	1319041	28:52	N	N

The violation in `aarddict` app occurs between the initialization of the activity (initializes the UI components and starts the dictionary service to load the dictionaries) and an event handler to lookup a word. The lookup cannot retrieve the requested word if the service gets initialized after the lookup. The pivot of the serializability violation is a write access to a variable `dictionaryService` in an asynchronous procedure invoked on the main thread that conflicts with the asynchronous procedure invoked on a background thread by the second event handler. We detected an event serializability violation in the `bookworm` app between the events dealing with user inputs to search for a book and navigating back to the previous screen. In this violation, while the first event handler performs the search in the background and not yet updated the `currSearchTerm` variable, the second event handler saves the stale `currSearchTerm` value in the cache. The pivot of the violation is a write access to the current search term in an asynchronous procedure invoked on the background thread. A violation detected in the `irccloud` app is presented in Sect. 2, which causes the app to send wrong messages. The pivot is a read access to the message text in an asynchronous procedure invoked on a background thread that conflicts with a write access in the double click event. A similar violation occurs in another user input sequence where the user types some text after pressing the “send” key. In the `vlille` benchmark, the serializability violation in the first line occurs when the user removes an item from the favorites list while the items are being loaded. The app throws an exception when the removal in the second event handler interleaves with the asynchronous procedure in the background.

Avoidable False Alarms. In the `grtftfs` benchmark, the conflict-serializability violation is *not* a bug or a serializability violation. (Conflict-serializability is stronger than serializability.) This violation is triggered by making two queries one after another. In an execution where the second event handler overwrites the query before the first event handler reads it in the background, both asynchronous procedures end up performing the same, later search. While technically this is not a serializability violation, we believe it is worthwhile to report conflict-serializability violations to the programmer, because fixing them would lead to improved code.

Inter-related Events. Some event handlers intervene the execution of another event by design. For such inter-related events, the event-serializability violation might not be a bug. The `vlille` benchmark has such an example (the second row on the table). In this scenario, the user navigates back while the app is loading a list of items asynchronously in a background thread. The event handler for back navigation sets the `mCancelled` flag of the `AsyncTask`. If this flag is set, the first event handler does not invoke the `AsyncTask`'s asynchronous `onPostExecute` procedure. Our techniques can be modified to consider inter-related events and task cancellation, but we leave this for future work.

9 Related Work

The UI framework in Android has been the focus of much work. Most existing tools for detecting concurrency errors investigate race detection [8,21,25]. Race conditions are low-level symptoms for a much broader class of concurrent programs which are often not indicative of actual programming errors. In this paper, we attempt to characterize and detect higher-level concurrency errors in Android programs. Robustness violations are incomparable with data-race freedom violations. Data races do not generally imply cyclic data dependencies among events, and cyclic data dependencies do not imply data races: e.g., surrounding each individual memory access within a cycle by a common lock eliminates possible races, but preserves cycles. Furthermore, checking conflict robustness is fundamentally more efficient than checking for data race freedom. Conflict event serializability requires tracking events, while data race freedom requires tracking individual program actions like reads and writes, which greatly outnumber events. Moreover, conflict robustness reduces to reachability in *sequential* programs, yielding significantly lower asymptotic complexity.

Recent work [29] proposes a static analysis to detect "anomalies" in event driven programs, i.e. accesses to the same memory location by more than one event handlers. Since many events access shared memory locations, this approach produces many false alarms, but programs without anomalies are conflict-event serializable. The works in [23,24] refactor applications by moving long running jobs to asynchronous tasks and transform improperly-used asynchrony constructs into correct constructs. Ensuring transformed asynchronous tasks do not race with their callers lends support to our work as it guarantees event-determinism.

The works in [12–14,26] target exploring interesting subsets of executions and schedules for asynchronous programs, that offer a large coverage of the execution space. This is orthogonal to the focus of our paper which is to investigate correctness criteria.

Conflict serializability [27] has been introduced in the context of databases and since then used as a tractable approximation of atomicity. We use serializability to formalize the fact that event handlers behave as if they were executed in isolation, without interference from others. While in other uses of serializability the transactions are sequential, in our case a single invocation of an event handler consists of several asynchronous procedures that can interleave arbitrarily in between them. Farzan and Madhusudan [15,16] and Bouajjani et al. [10] investigate decision procedures for conflict serializability of finite-state concurrent models while checking serializability in general has been approached using both static, e.g., [18,20,32,34], and dynamic tools, e.g., [17,19,30,33].

Determinism has been largely advocated in the context of concurrent programs, e.g., [9,31], since it simplifies the debugging and verification process. Prior work has introduced static verification techniques, e.g., [22] but also dynamic analyses based on testing, e.g., [11,28]. Differently from prior work, we provide a methodology for checking determinism of event-driven asynchronous programs that ultimately reduces to a reachability problem in a sequential program.

References

1. https://github.com/irccloud/android/commit/c81f3374
2. http://github.com/irccloud/android/tree/9e2f5cf04e
3. F-Droid - Free and Open Source App Repository. http://f-droid.org/
4. Java pathfinder. http://babelfish.arc.nasa.gov/trac/jpf/
5. https://github.com/burcuku/async-robustness-checker
6. Alur, R., McMillan, K.L., Peled, D.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. **160**(1–2), 167–188 (2000)
7. Arzt, S., Rasthofer, S., Bodden, E.: Instrumenting android and Java applications as easy as abc. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 364–381. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40787-1_26
8. Bielik, P., Raychev, V., Vechev. M.: Scalable race detection for android applications. In: Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, NY, USA, pp. 332–348. ACM (2015)
9. Bocchino, Jr. R.L., Adve, V.S., Adve, S.V., Snir, M.: Parallel programming must be deterministic by default. In: Proceedings of 1st USENIX Conference on Hot Topics in Parallelism, HotPar 2009, CA, USA (2009)
10. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 290–309. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37036-6_17
11. Burnim, J., Sen, K.: Asserting and checking determinism for multithreaded programs. Commun. ACM **53**(6), 97–105 (2010)
12. Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. SIGPLAN Not. **46**(1), 411–422 (2011). ISSN 0362-1340
13. Emmi, M., Lal, A., Qadeer, S.: Asynchronous programs with prioritized task-buffers. In: Proceedings of International Symposium on Foundations of Software Engineering, FSE 2012, pp. 48:1–48:11. ACM (2012)
14. Emmi, M., Ozkan, B.K., Tasiran, S.: Exploiting synchronization in the analysis of shared-memory asynchronous programs. In: Proceedings of International SPIN Symposium on Model Checking of Software, pp. 20–29. ACM (2014)
15. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008). doi:10.1007/978-3-540-70545-1_8

16. Farzan, A., Madhusudan, P.: The complexity of predicting atomicity violations. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 155–169. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00768-2_14
17. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. Sci. Comput. Program. **71**(2), 89–109 (2008)
18. Flanagan, C., Freund, S.N., Lifshin, M., Qadeer, S.: Types for atomicity: static checking and inference for Java. ACM Trans. Program. Lang. Syst. **30**(4), 20 (2008)
19. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 293–303 (2008)
20. Hatcliff, J., Robby, Dwyer, M.B.: Verifying atomicity specifications for concurrent object-oriented software using model-checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 175–190. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24622-0_16
21. Hsiao, C.-H., Yu, J., Narayanasamy, S., Kong, Z., Pereira, C.L., Pokam, G.A., Chen, P.M., Flinn, J.: Race detection for event-driven mobile applications. In: Proceedings of 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 326–336. ACM (2014)
22. Bocchino Jr. R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H. and Vakilian, M.: A type and effect system for deterministic parallel Java. In: Proceedings of OOPSLA, pp. 97–116 (2009)
23. Lin, Y., Ra, C., Dig, D.: Retrofitting concurrency for android applications through refactoring. In: Proceedings of International Symposium on Foundations of Software Engineering, FSE 2014, NY, USA, pp. 341–352. ACM (2014)
24. Lin, Y., Okur, S., Dig, D.: Study and refactoring of android asynchronous programming. In: Proceedings of ASE (2015)
25. Maiya, P., Kanade, A., Majumdar, R.: Race detection for android applications. In: Proceedings of 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 316–325. ACM (2014)
26. Ozkan, B.K., Emmi, M., Tasiran, S.: Systematic asynchrony bug exploration for android apps. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, pp. 455–461 (2015)
27. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)
28. Sadowski, C., Freund, S.N., Flanagan, C.: SingleTrack: a dynamic determinism checker for multithreaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 394–409. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00590-9_28
29. Safi, G., Shahbazian, A., Halfond, W.G.J., Medvidovic, N.: Detecting event anomalies in event-based systems. In: Proceedings of International Symposium on Foundations of Software Engineering FSE, pp. 25–37. ACM (2015)
30. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predicting serializability violations: SMT-based search vs. DPOR-based search. In: Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, Revised Selected Papers, pp. 95–114 (2011)
31. Steele, Jr. G.L.: Making asynchronous parallelism safe for the world. In: Proceedings of 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1990, NY, USA, pp. 218–231. ACM (1990)
32. von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. J. Object Technol. **3**(6), 103–122 (2004)

33. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Softw. Eng. **32**(2), 93–110 (2006)
34. Yi, J., Disney, T., Freund, S.N., Flanagan, C.: Cooperative types for controlling thread interference in Java. In: International Symposium on Software Testing and Analysis, ISSTA, pp. 232–242 (2012)