Abstract

Android is a popular mobile operating system, providing a rich ecosystem for the development of applications which run on the Android platform. Entities such as the device user, network and sensors interact continuously with the mobile device causing an Android application to face a continuous stream of input (events). In spite of having to perpetually handle a huge volume of events, Android applications are expected to be responsive to new events from the environment. The Android framework achieves this by exposing applications to a concurrency model which combines multi-threading with asynchronous event-based dispatch. While this enables development of efficient applications, unforeseen thread interleavings combined with non-deterministic ordering of events can lead to subtle concurrency bugs in Android applications.

In an Android application, threads communicate through shared variables and by sending events to each other's event queues. Even though typically a thread processes events in its event queue and invokes corresponding event handlers in a FIFO order, this ordering however can be altered by attributes such as delays and priorities associated with the events. The existing literature extensively describes techniques to detect concurrency bugs such as data races, deadlocks and atomicity violations in multi-threaded programs. Recent works also present techniques to analyze bugs manifested due to single-threaded event-driven concurrency. However, the complex event-driven semantics of Android applications combined with the thread-based semantics render a naïve extension of such concurrency analysis techniques either less effective or unsound for Android applications. This thesis takes the initial steps towards developing data structures and algorithms to facilitate effective dynamic concurrency analysis of Android applications.

We firstly formalize the concurrency behaviour of Android applications by giving concurrency semantics. Using this semantics we derive a set of rules to reason about the *happens-before* ordering between operations in an Android execution trace. These rules induce a partial order called a happens-before (HB) relation on the operations of an execution trace. Our HB relation generalizes the so far independently studied HB relations for multi-threaded programs and single-threaded event-driven programs. We have developed a happens-before based dynamic race detector for Android applications, called DroidRacer, which detects data races across

threads as well as race conditions between memory accesses from different event handlers on the same thread. DroidRacer has detected several races in various Android applications.

We identify a major bottleneck in the computation of the HB relation for Android applications, that of discovering HB order between event handlers executed on the same thread. HB order between events in Android is characterized by complex HB rules which order a pair of event handlers by inspecting, for example, the order between operations which enqueued the corresponding events. Android applications usually receive a large number of events even within a short span of time, which increases the cost of evaluating such HB rules. As a result HB computation using standard data structures such as vector clocks alone becomes inefficient in case of Android applications. We propose a novel data structure, called *event graph*, that maintains a subset of the HB relation to efficiently infer order between any pair of events. We present an algorithm, called EventTrack, which improves efficiency of vector clock based HB computation for event-driven programs using event graphs. Evaluation of EventTrack against a state-of-the-art HB computation technique for Android applications, yielded significant speedup.

The scope of the happens-before based dynamic race detector is limited to the thread interleaving corresponding to the execution trace inspected. However, a systematic state space exploration technique such as a stateless model checker can explore all the thread interleavings, and also identify harmful manifestations of data races which may happen only when multiple racing memory accesses are performed in a particular order. Partial order reduction (POR) is a technique used by stateless model checkers to reduce the state space to be explored while preserving certain interesting program properties. The existing POR techniques selectively explore different thread interleavings only to reorder pairs of racing operations from different threads. However, they fail to follow this strategy for events and hence end up eagerly exploring all possible ordering between event handlers executed on the same thread, even if reordering them does not lead to different states. We propose a new POR technique based on a novel backtracking set called the *dependence-covering set*. Events handled by the same thread are reordered by our POR technique only if necessary. We prove that exploring dependence-covering sets suffices to detect all deadlock cycles and assertion violations in the non-reduced state space. On execution traces obtained from a few Android applications, we demonstrate that our technique explores many fewer transitions —often orders of magnitude fewer— compared to exploration based on persistent sets, a popular POR technique which explores all possible orderings between events.

The tools, (i) DroidRacer, a race detector for Android applications, (ii) a standalone implementation of EventTrack, and (iii) EM-Explorer – a proof-of-concept model checking framework which simulates the non-deterministic behaviour exhibited by Android applications, developed as part of this thesis have been made public.