2019 IEEE 3rd Information Technology,Networking,Electronic and Automation Control Conference (ITNEC 2019)

Verifying nondeterministic processes driven by broadcasts on Android

Chu Luo
The University of Melbourne
Victoria, Australia
Email: CHUL3@student.unimelb.edu.au

Xu Ma, Yanshan Tian
Ningxia Normal University
NingXia, China

Jorge Goncalves, Eduardo Velloso, Vassilis Kostakos
The University of Melbourne
Victoria, Australia

***Abstract*—Broadcasts in Android facilitate inner-process and inter-process communications. Although broadcasts enable high scalability and loose coupling in achieving collaboration among system components, there is no support for developers to verify the correctness of broadcast-driven nondeterministic processes. To overcome this challenge, we propose a verification approach for systems driven by all the four types of Android broadcasts. Our approach uses the PROMELA language to model broadcast senders and receivers, with regards to unique features of each broadcast type. Based on our design of initialisation procedures, developers can verify properties of their systems using the SPIN model checker. We evaluate our approach in a case study on an example system. Results show that our approach can effectively conduct verification in terms of safety, liveness and never claims with limited computing resources.**

I. Introduction

Android offers powerful mechanisms of broadcasts to enable inner-process and inter-process communications. The components of an Android application can send broadcast messages to components of the same application, as well as to components of external applications. Also, the Android operating system (OS) uses broadcasts to send system-level events (e.g., a battery-low message) to all installed applications. Similar to the publish-subscribe design pattern, the implementation of broadcasts is very simple on Android. The sender can call the broadcast API to send a broadcast message, without knowing the potential receivers. To subscribe a type of broadcasts, application components only have to register a permission in Android *Manifest* to specify the details of the broadcasts. If a broadcast arrives, the receiver of the application component can immediately respond, without knowing the sender. Since broadcasts enable high scalability and loose coupling of systems involving collaborations and information sharing, Android developers have built a significant number of broadcast-based applications, such as context management middleware (e.g., [1]) and context-aware services.

Despite the benefits brought by broadcasts, it is a challenging task to verify the correctness of programs driven by broadcasts. Often, developers of mobile applications rely on specialised simulation tools to conduct data-driven tests by replaying historical data (e.g., [2]) or generating bogus data (e.g., [3]) as tests cases. These tools support various data types including sensory data and contextual events (e.g., all kinds of broadcasts). Some tools and approaches, such as MobiPlay [4], [5] and TestAWARE [6], provide the context replay function allowing testers to test context-aware applications. Although these tools are effective in the detection and reproduction of bugs within deterministic processes, several studies [2], [7] point out that these tools can hardly support testers to examine the nondeterministic processes (e.g., I/O, thread scheduling, resource allocation and communications between concurrent components) in mobile applications. This is because nondeterministic processes may enter many different states and generate various outputs each time they respond to the same input.

Regarding nondeterministic processes, model checking is a popular and effective technique to examine their correctness in practice [8]. Compared to data-driven tests, model checking approaches operate on a mathematical abstraction of a system (e.g., finite-state machines). These approaches perform an exhaustive search of all possible system states to detect violations of correctness properties such as absence of deadlocks. For mobile applications, there exist several model checking techniques, such as JPF-Android [7] and the generation technique of library models [9]. However, these model checking techniques are effective only in verifying internal behaviours of a standalone applcation without communication protocols (e.g., broadcasts). Developers cannot rely on them to verify programs driven by broadcasts or other communications.

To bridge this gap, we propose an approach to enable the verification of broadcast-driven programs with nondeterministic processes on Android. Unlike the second most popular mobile platform iOS banning any direct inter-process communication, the 4 types of broadcasts on Android have high richness and representativeness of different broadcasting features on mobile devices. Based on our analysis on each type of them, we present the modelling method to abstract dynamic behaviours of the sender and receiver using the PROMELA language [10]. This model method is able to express the unique features of each broadcast type. Depending on actual software specifications, developers can add component-specific behaviours into the sender and different receivers. To conduct model checking, we choose the SPIN [11] model checker due to its popularity and verification support for multiple correctness properties. For each broadcast type, we design the initialisation process with which developers can launch an exhaustive search on SPIN to verify various correctness

978-1-5386-6243-4/19/$31.00 ©2019 IEEE

properties of their applications. We evaluate our approach by conducting a case study on an example system. The results highlight the effectiveness of our approach in verification in terms of safety, liveness and never claims with limited computing resources, e.g., a common PC.

II. Android Broadcast Overview

On the Android platform, application components can deliver broadcast messages to the same application and/or to external applications. Application components can receive broadcast messages from the operating system, the application itself and/or other Android applications. The mechanism of Android broadcasts is similar to the publish-subscribe design pattern, where publishers inject messages into classes without knowledge about subscribers, and subscribers receive interested messages without knowing publishers. Android broadcasts enable loose coupling for the development of applications with internal and inter-process communications, improving the efficiency of data sharing. On Android smartphones, typical publishers of Android broadcasts can be context recognition applications (e.g., localisation applications), and subscribers can be services adapting to context (e.g., location-based restaurant recommendation applications).

In contrast, the second most popular mobile platform iOS has a strict sandbox policy since version 7.0. This policy does not allow any direct inter-process communication. Although secure, iOS has very limited capability for applications to collaborate. Hence, we choose Android due to its richness and representativeness of broadcasting on mobile devices.

Android provides four types of broadcasts: normal broadcasts (API: sendBroadcast), local broadcasts (API: LocalBroadcastManager.sendBroadcast), ordered broadcasts (API: sendOrderedBroadcast) and sticky broadcast (API: sendStickyBroadcast). Particularly, the only difference between local broadcasts and normal broadcasts is that local broadcasts cannot be received outside the application of the sender component. Hence, we combine these two types of broadcasts in our analysis and verification design.

A. Normal/Local Broadcasts

Normal broadcasts and local broadcasts are sent to all receivers subscribing to the publishers in the global or local scope. Normal broadcasts from an application can be received by a receiver inside or outside the application. Comparatively, local broadcasts can only be received inside the application of the publisher.

Normal broadcasts and local broadcasts are extremely widely used: e.g., normal broadcasts used by 379,742 times, local broadcasts used by 46,428 times, within all open-source software projects on the GitHub platform (measured by its search engine). However, it is challenging to verify the correctness of applications driven by them. They represent typical nondeterministic events on Android and other similar mobile platforms, because they result in a random sequence where each receiver obtains the message.

For example, Fig. 1 depicts a normal/local broadcast sent from P to all receivers Q_1, Q_2, Q_3 in the valid scope. Any one of transmissions $\{P \rightarrow Q_1,\ P \rightarrow Q_2,\ P \rightarrow Q_3\}$ can happen before, between or after the other two. After the communication, each receiver may concurrently perform its operations. Meanwhile, the sender component may also continue running.

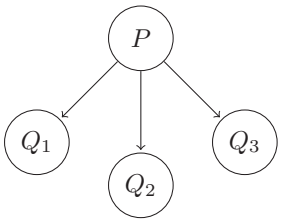

Fig. 1. An example of normal/local broadcasts: sender P sends a broadcast to all receivers Q_1, Q_2, Q_3.

Hence, given n receivers with q_i states and a sender with p states after the communication, the number of all possible states during and after the communication is

$$S = p \times n! \times \prod_{i=1}^{n} q_i. \tag{1}$$

Exhaustive checking such nondeterministic processes is challenging for common testing tools and techniques, but suitable for model checkers.

B. Ordered Broadcasts

Ordered broadcasts are also commonly used: 39,942 times of usage within all open-source projects on GitHub. Ordered broadcasts have two features beyond normal broadcasts:

1) **Priority**: receivers of ordered broadcasts can be registered with a priority. Ordered broadcasts are first sent to receivers with high priorities, then to receivers with lower priorities, and finally to receivers without priorities. If a group of receivers have the same priorities, they will receive the broadcast in a random order.
2) **Termination**: with the termination feature, ordered broadcasts also allow the receiver to decide the further transmission. That is, after a receiver obtains an ordered broadcast, it can terminate the further transmission of the broadcast to other receivers with lower priorities using the API *abortBroadcast()*. However, if a receiver terminates the broadcast, receivers with the same priority can still receive the broadcast later than the termination.

To illustrate the features of ordered broadcasts, we show an example in Fig. 2. Sender P sends an ordered broadcast to all receivers Q_1, Q_2, Q_3, where Q_1 and Q_2 have equally high priority and Q_3 has lower priority. Either $P \rightarrow Q_1$ or $P \rightarrow Q_2$ can happen before the other. Q_1 is implemented to terminate the broadcast when it obtains the broadcast. Suppose $P \rightarrow Q_1$ happens first, even if Q_1 terminates the broadcast, $P \rightarrow Q_2$ will happen. However, in any case, $P \rightarrow Q_3$ never happens.

Thus, it is necessary to model the priority and termination feature, as well as the nondeterministic order of receivers with

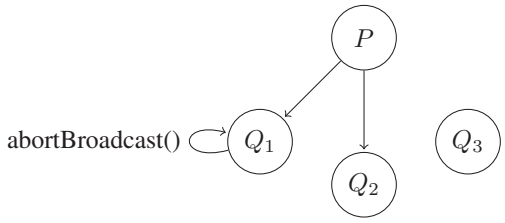

Fig. 2. An example of ordered broadcasts: sender P sends a broadcast to all receivers Q_1, Q_2, Q_3, where Q_1 and Q_2 have equally high priority and Q_3 has lower priority.

the same priority, for the verification of Android applications driven by ordered broadcasts.

C. Sticky Broadcasts

Sticky broadcasts are the least popular type of broadcasts: 21,757 times within all open-source projects on GitHub. Unlike other kinds of broadcasts, sticky broadcasts remain effective to newly activated receivers after the moment of transmission, similar to a browser cookie. For example, a localisation application on a smartphone sends a sticky broadcast of the current location. Some seconds later, the phone user opens a location-based service application with a receiver of the sticky broadcast. Although just initialised, the receiver of this location-based service application can still receive the previous sticky broadcast to obtain the latest location. If a sticky broadcast is no longer useful, the sender or receiver can remove it using API $removeStickyBroadcast()$.

To verify programs with sticky broadcasts, besides the nondeterministic receiving order implied by any broadcast, the model should include the removal feature and the possible existence of receivers which are created after the broadcast.

III. Verification

Considering that broadcasts are often transmitted among different applications (source code of all applications are not always available), we argue that JPF is not suitable to verify them, due to its source code requirement and inability of communication modelling. Instead, we demonstrate the verification techniques for each using the PROMELA language and SPIN model checker, due to their capabilities of abstracting and verifying programs with communication protocols. This way does not need any internal implementation detail. Developers only need to have abstractions of the dynamic program behaviours. For each type of broadcasts, we model the processes of senders and receivers using PROMELA with two goals:

1) reflecting the features of the type of broadcasts using the communication functions provided by PROMELA;
2) constructing a loop in the PROMELA model so that the SPIN model checker can automate the exhaustive searching of all possible states.

Also, we provide the details required for the initialisation of verification with the PROMELA models.

A. Normal/Local Broadcasts

1) Senders: Algorithm 1 shows our PROMELA design of a sender of normal and local broadcasts, which contains a loop of all its actions for the SPIN model checker to exhaustively search all the states of the sender and receivers. We use the message passing channel type *chan* of PROMELA to abstract a single broadcast from the sender to a receiver. Since Android sends broadcasts at the same time, we use an atomic action to model the whole sending process. However, broadcasts are asynchronous for receivers to receive. Thus, we set the capacity of channels $ch[T]$ to 1 during the initialisation.

Algorithm 1 Sender of normal and local broadcasts

Input: Number of receivers T; channels of each receiver $ch[T]$ with channel capacity 1; readiness of each receiver $listening[T]$; broadcast message M;
Output: Verification results from SPIN model checker

```
1:  loop
2:    atomic action{
3:    let a be ⋀_{i=1}^{T} listening[T];
4:    if a is true then
5:      for all channel ch[i], i = 1 to T do
6:        send M to ch[i];
7:        set listening[i] to false;
8:      end for
9:    else
10:     go to step 1;
11:   end if
12:   }end of atomic action
13:   do planned computation after the communication;
14: end loop
```

If the channel type *chan* obtains more messages than its capacity, users have to specify whether SPIN should drop the extra messages. On Android, we found that receivers receive broadcasts from the FIFO queue. A new incoming broadcast cannot result in the loss of an old broadcast that is not yet received by the receiver. Hence, dropping messages is not realistic. We decide to block the sender before all receivers process the previous broadcast. To express this, we include readiness states of each receiver by $listening[T]$. Before sending a new broadcast, the sender checks all the readiness states (line 3 and 4). If a broadcast is not read by all receivers, the sender will wait (line 9 and 10). After sending a broadcast, the sender sets all the readiness states to false (line 7). Then, the sender starts to do planned computation after the communication (line 13), depending on actual software design.

2) Receivers: Comparatively, the design of receivers listening to normal and local broadcasts is simple. The implementation using PROMELA should adopt the code shown in Listing 1. Similar to the sender, receivers also have a loop to support exhaustive search. To check the invalid ending states, we put an *end* label in the beginning of the loop (line 3). Once a receiver receives a broadcast (line 4), it starts its concurrent actions which are specified by developers according to the

software design (line 5). In the end of actions, it should report its readiness of receiving a new broadcast (line 6). Otherwise, the sender will wait forever. The concurrent actions should be easy for developers to specify in PROMELA. In practice, the receivers react to broadcasts very quickly. Android OS detects and forcibly finishes any long time process in receivers.

```
1 proctype receiverThread(int ID){
2   int Msg;
3 end: do
4        ::listener.ch[ID]?Msg->
5             //concurrent actions
6             listening[ID]=true;
7        od
8 }
```

Listing 1. Receiver of normal and local broadcasts

3) Initialisation: During verification, the initialisation sequence of the sender and receivers is the following:

1) setting readiness states of each receiver to true;
2) starting all the sender and receivers.

B. Ordered Broadcasts

1) Senders: To express the priority and termination feature of the sender of ordered broadcasts in the PROMELA model, we propose Algorithm 2. Similar to Algorithm 1, it also has a main loop supporting exhaustive search. For the termination feature, Algorithm 2 simply uses a boolean signal A. For the priority feature, Algorithm 2 takes as input a constant array (i.e., $priority[T]$) of all the priority values from each receiver. To send broadcasts according to the priority order, we use a runtime (i.e., dynamic) priority array $priorityRuntime[T]$ that is initialised in the first loop using $priority[T]$ (line 3-5). After the initialisation, the second loop sends broadcasts with an atomic round-by-round procedure. This procedure first detects and handles the termination request from receivers (line 8, 9). Then it identifies the active receivers with the highest priority and sends broadcasts to them (line 11-23). During the transmission, it marks the targeted receivers as inactive (line 20), by setting their runtime priorities in $priorityRuntime[T]$ to the integer B, which is smaller than any possible priority value. This is to ensure that transmissions of the same broadcast will not repeat connections to the receivers with highest priority in the constant array $priority[T]$. When all receivers receive the broadcasts, the procedure finishes one round (line 14, 15). Developers can implement the application-specific computation which the sender should do after the communication (line 27).

2) Receivers: The receivers of ordered broadcasts are able to send termination request to the sender. This feature can be simply implemented by setting the boolean termination signal A as true. Except this feature, the receivers of ordered broadcasts are identical to receivers of normal/local broadcasts (Listing 1). Developers can add the concurrent behaviours of software specifications into receivers.

Algorithm 2 Sender of ordered broadcasts

Input: Number of receivers T; channels of each receiver $ch[T]$ with channel capacity 1; readiness of each receiver $listening[T]$; broadcast message M; priorities of each receiver $priority[T]$; termination signal A; receiver smaller than any possible priority value B;

Output: Verification results from SPIN model checker

```
1: loop
2:   set termination signal A to false;
3:   atomic action{
4:   copy each value of priority[T] into a runtime priority
     sequence priorityRuntime[T];
5:   }end of atomic action
6:   loop
7:     atomic action{
8:     if A is true then
9:       go to step 27;
10:    else
11:      let a be ⋀_{i=1}^{T} listening[T];
12:      if a is true then
13:        find the maximum maxPriorityRuntime in
           priorityRuntime[T];
14:        if maxPriorityRuntime == B then
15:          go to step 27;
16:        else
17:          for all maxPriorityRuntime ==
             priorityRuntime[i], i = 1 to T do
18:            send M to ch[i];
19:            set listening[i] to false;
20:            set priorityRuntime[i] to B;
21:          end for
22:        end if
23:      end if
24:    end if
25:    }end of atomic action
26:  end loop
27:  do planned computation after the communication;
28: end loop
```

3) Initialisation: To perform a verification using the model of ordered broadcasts, the initialisation sequence contains the following steps:

1) assigning $priority[T]$ to the actual priority values of each receiver on Android;
2) setting the boolean termination signal A as false;
3) setting the B smaller than any value in $priority[T]$;
4) setting readiness states of each receiver $listening[T]$ to true;
5) starting all the sender and receivers.

C. Sticky Broadcasts

1) Senders: Algorithm 3 gives our model of a sender delivering sticky broadcasts. Due to the similarity of mechanisms, it has all the elements of Algorithm 1. Besides, it also

Algorithm 3 Sender of sticky broadcasts

Input: Number of receivers T; channels of each receiver $ch[T]$ with channel capacity 1; readiness of each receiver $listening[T]$; Creation statuses of each receiver $created[T]$; broadcast message M;

Output: Verification results from SPIN model checker

1: **loop**
2: **atomic action**{
3: let a be $\bigwedge_{i=1}^{T} listening[T]$;
4: **if** a is true **then**
5: **for all** channel $ch[i]$, $i = 1$ to T **do**
6: copy each value of $created[T]$ into a runtime creation status array $createdRuntime[T]$;
7: send M to $ch[i]$;
8: set $listening[i]$ to false;
9: **end for**
10: **else**
11: go to step 1;
12: **end if**
13: }**end of atomic action**
14: do planned computation after the communication;
15: **end loop**

takes as input an array of creation statuses of all receivers $created[T]$. This array describes whether each receiver is created at the moment of broadcasting. To perform dynamic creations of receivers, we use a runtime creation status array $createdRuntime[T]$. $createdRuntime[T]$ is initialised with the values in $created[T]$ in the beginning of each round of exhaustive search (line 6). When the sender delivers a sticky broadcast, it injects the message to the channels of all receivers, even including receivers which are not yet created. Restricted by $createdRuntime[T]$, the receivers which are not yet created cannot obtain the message from their channels. After broadcasting, the model of created receivers may trigger the creation of new receivers by modifying the values in $createdRuntime[T]$. Then, late created receivers can obtain the message which already exists in their channel.

2) Receivers: The receiver model of sticky broadcasts is more complex than those of the other broadcasts. Hence, we present the design of receiving the messages and the designs of its features individually.

Listing 2 shows the mechanism of receiving sticky broadcasts in our PROMELA model, which also includes a loop of endless receiving attempts to support exhaustive search by SPIN. To check invalid ending states, an *end* label is added in the beginning of the main loop (line 3). To check the existence of sticky broadcasts, we use the channel poll operation provided by PROMELA (line 8). When a sticky broadcast is available, a receiver must confirm its creation status before it obtains the message (line 9).

```
1  proctype receiverThread(int ID){
2    int Msg;
3  end: do
4    ::true->
5  L0:    skip;
6         atomic{
7         if
8           ::(listener.ch[ID]?[Msg])&&
9            (runtimeCreated[ID]==true)->
10             listener.ch[ID]?Msg;
11          ::else->goto L0;
12        fi}
13        //concurrent actions
14        listening[ID]=true;
15   od}
```

Listing 2. Receiver of sticky broadcasts

Besides receiving the messages, the receiver model of sticky broadcasts should express the following features:

1) creating new receivers so that they may receive the existing sticky broadcast;
2) removing the existing sticky broadcast to stop late created receivers from receiving it.

The first feature can be implemented by setting the runtime creation status of a receiver to true, for example, receiver 1 creates receiver 2:

```
1  atomic{
2   if
3       ::(ID==1)->
4           runtimeCreated[2]=true;
5       ::else->skip;
6   fi}
```

Listing 3. Creating new receivers

The second feature can be achieved by popping the messages from all channels which are not empty. If a message is successfully removed from a channel, the readiness of the related receiver should be set to true in $listening[T]$, so that the sender can send next broadcast for SPIN to perform another round of search. Listing 4 gives an example of the implementation.

3) Initialisation: The initialisation steps for verifying programs with sticky broadcasts are the following:

1) assigning creation statuses of each receiver $created[T]$ to the actual states before broadcasting;
2) setting all values of runtime creation status array $createdRuntime[T]$ as false;
3) setting readiness states of each receiver $listening[T]$ to true;
4) starting all the sender and receivers, including receivers which actually do not exist before broadcasting (because their creation statuses are represented by $created[T]$).

```
1  atomic{
2   int i;
3   for (i : 0 .. T-1) {
4    if
```

TABLE I
MODEL OF THE SYSTEM

Component Name	Sender/Receiver	Receiver Priority
Application P	Sender	N/A
Application $Q1$	Receiver	3
Application $Q2$	Receiver	2
Application $Q3$	Receiver	2

```
5       ::(nempty(listener.ch[i]))->
6           int Msg;
7           listener.ch[i]?Msg;
8           listening[i]=true;
9       ::(empty(listener.ch[i]))->skip;
10    fi}
11  }
```

Listing 4. Removing sticky broadcasts

IV. CASE STUDY

In this section, we present a case study on an example system to show the effectiveness of our approach. The example system is a real-world Android system driven by ordered broadcasts. We choose ordered broadcasts for three reasons:

1) The features of normal/local broadcasts can be reflected by these two other kinds of broadcasts.
2) The termination feature of ordered broadcasts is similar to the removal feature of sticky broadcasts.
3) The priority feature of ordered broadcasts is unique among all kinds of broadcasts.

This system consists of 4 Android applications with communications via ordered broadcasts. 3 serve as receivers listening to broadcasts sent by the other application. This system structure is typical in scenarios where many services rely on a data collection middleware. Table I summarises the features of 4 Android applications. We abstract this system into a PROMELA model. To illustrate verification of each property, we may have to slightly modify the model (e.g., adding an assertion).

A. Safety

SPIN examines safety in two aspects: invalid end states (i.e., deadlocks) and assertions. To check invalid end states, an *end* label should be added in the beginning of the main loop during the implementation of senders (line 1 of Algorithm 1-3). Assertions can be added anywhere to check the properties of models, depending on the actual specifications of software.

For example, in the receivers of the model, to confirm that $Q1$ always gets the message first due to its highest priority, we added an assertion *assert(abort==false)* when $Q1$ receives the message, and we applied a termination *abort=true* after this assertion. We also applied the termination after the broadcast is obtained by other receivers. The result of verification is shown in Listing 5. With default settings, SPIN did not detect any deadlock or failure of assertions.

```
1  Full statespace search for:
2  never claim - (not selected)
3  assertion violations +
4  cycle checks - (disabled by -DSAFETY)
5  invalid end states +
6  State-vector 136 byte,
7  depth reached 122, errors: 0
```

Listing 5. Safety verification result 1

Next, we show a case of an failed assertion. We removed all terminations and assertions in the model. We applied a termination *abort=true* when $Q2$ receives the message before $Q3$. We added an assertion *assert(abort==false)* when $Q3$ receives the broadcast. Listing 6 gives the result of verification. SPIN reported a violated assertion. This is because $Q2$ and $Q3$ have equal priority so that $Q3$ can still receives the broadcast after the termination by $Q2$.

```
1  assertion violated (abort==0)
2                     (at depth 136)
3  Full statespace search for:
4  never claim - (not selected)
5  assertion violations +
6  cycle checks - (disabled by -DSAFETY)
7  invalid end states +
8  State-vector 136 byte,
9  depth reached 211, errors: 1
```

Listing 6. Safety verification result 2

B. Liveness

Absence of deadlocks cannot ensure the liveness properties of programs. Hence, SPIN supports the verification of liveness by detecting two kinds of undesirable cycles: non-progress cycles and acceptance cycles. A non-progress cycle represents a circular sequence where the program is not making effective progress. An acceptance cycle is a circular sequence where an accepting state happens infinitely often.

To show a simple case, we added an action *action=1*, where *action* is shared integer across receivers, with the progress label into each receiver as their concurrent actions after the broadcast communication. With the weak fairness constraint on the liveness verification mode, SPIN found no non-progress cycles (Listing 7). It is worth nothing that, without the weak fairness constraint, SPIN would report a non-progress cycle in the sender. This is because the sender must wait for each receiver to process last broadcast. If there is no weak fairness constraint, the execution of checking readiness in the sender will remain forever, stopping receivers from proceeding.

```
1  Full statespace search for:
2  never claim + (:np_:)
3  assertion violations +
4          (if within scope of claim)
5  non-progress cycles +
6          (fairness enabled)
```

```
invalid end states -
        (disabled by never claim)
State-vector 140 byte,
depth reached 289, errors: 0
```

Listing 7. Liveness verification result 1

Next, we show an example case causing a non-progress cycle. Towards the action *action=1*, we added a condition *(1==0)* which is always false. Then the program can never enter the progress-labelled action *action=1*. SPIN reported the non-progress cycle (Listing 8).

```
non-progress cycle (at depth 327)
Full statespace search for:
never claim + (:np_:)
assertion violations +
        (if within scope of claim)
non-progress cycles +
        (fairness enabled)
invalid end states -
        (disabled by never claim)
State-vector 140 byte,
depth reached 609, errors: 1
```

Listing 8. Liveness verification result 2

Then, to verify the same code using the detection of acceptance cycles, we replaced the progress label with an accept label. Now the program can never enter the accept-labelled action *action=1*, meaning that there is no liveness violation. SPIN reflected this fact (Listing 9).

```
Full statespace search for:
never claim - (not selected)
assertion violations +
acceptance cycles + (fairness enabled)
invalid end states +
State-vector 136 byte,
depth reached 258, errors: 0
```

Listing 9. Liveness verification result 3

To make an acceptance cycle, we removed the condition *(1==0)*, so that the accept-labelled action *action=1* is infinitely often executed. However, we failed to launch an exhaustive search of acceptance cycles using our PC due to the limited 8 GB of physical memory. The required maximal search depth was extremely large. Hence, we argue that developers of broadcast-based systems should attempt to verify liveness properties using the detection of non-progress cycles. Theoretically, searching for a non-progress cycles is easier than an acceptance cycle, since the area containing progress-labelled code can be removed from the search (it will not be a part of non-progress cycle). In contrast, if an acceptance cycle exists, the area containing accept-labelled code must be a part of it, possibly resulting in an extremely large search depth.

C. Never Claims and LTL

Never claims can be used to describe a behaviour or a state that should never emerge in the model. LTL can also be adopted for this goal. Given a LTL formula, SPIN can automatically generate a never claim to conduct verification. Since they are equivalent, we choose LTL due to its simplicity.

```
Full statespace search for:
never claim + (p)
assertion violations +
        (if within scope of claim)
acceptance cycles +
        (fairness enabled)
invalid end states -
        (disabled by never claim)
State-vector 144 byte,
depth reached 327, errors: 0
```

Listing 10. LTL verification result 1

```
acceptance cycle (at depth 476)
never claim + (p)
assertion violations +
        (if within scope of claim)
acceptance cycles +
        (fairness enabled)
invalid end states -
        (disabled by never claim)
State-vector 144 byte,
depth reached 1002, errors: 1
```

Listing 11. LTL verification result 2

As an example case, we added an action *action=1* in $Q1$ after it receives the message, and we added another action *action=0* in $Q2$ after its communication. For the verification, We used the LTL formula:

$$ltl\ p\ \{\ []\ ((action == 1) -> <>\ (action == 0))\}. \quad (2)$$

It means that every time *action==1* happens, eventually *action==0* will happen afterwards. With the default settings for LTL-based verification (SPIN requires the acceptance cycle detection mode to launch an exhaustive search), SPIN reported no violation of this LTL (Listing 10).

To show a violation of this LTL, we simply removed the action *action=0* in $Q2$ after it receives the message. Then, SPIN detected a violation via an acceptance cycle which means that there exists a loop with the state *action==1* but without the state *action==0* (Listing 11).

V. Discussion

A. Usage Scenarios

Programs driven by broadcasts on Android, including applications designed for smartphones, smartwatches and other hardware with an Android OS, can be verified using our approach. The senders of broadcasts can be developers' applications as well as Android OS itself. Usually, senders are middleware clients which periodically sending contextual events such as software notifications and sensory data. Senders can also be specialised applications sharing contextual data collected from external devices or sensors, for example, a system sharing data

received from various Device-to-Device communications [12], [13], [14] or a near-infrared spectroscopy system connected to an Android smartphone [15].

Although our approach aims at the Android platform, it is not limited to Android applications only. Application developers can use our design of verification to verify any program having the same mechanisms of broadcasting. For instance, this kind of broadcasts can be essential techniques in web services [16]. Another usage scenario is smart infrastructure with the Internet of Things, such as [17] and [18], where various devices and smart objects communicate and collaborate together using broadcasts. Developers can use our approach to verify the properties of such component-based and broadcast-driven systems.

VI. Conclusion and Future Work

Broadcasts are largely adopted by Android applications for inner-process and inter-process communications. Although broadcasts enable high scalability and loose coupling in the development of collaboration among system components, it is challenging to verify the correctness of nondeterministic events triggered by broadcasts. To address this issue, we propose the verification techniques for systems driven by the 4 types of broadcasts provided by Android. For each type of broadcasts, we present the modelling method to abstract the dynamic behaviours of the sender and receiver with their unique features using the PROMELA language. Based on our design of initialisation processes for each broadcast type, application developers can use the SPIN model checker to exhaustively search all the possible states in the PROMELA model for the verification of correctness properties. We conduct a case study with an example system. Results show that our approach can effectively conduct verification on safety, liveness and never claims with limited computing resources.

Our future work plans to provide basic automation support for the modelling process by parsing the source code of the sender and receivers related to certain broadcasts. We also plan to build a tool to generate PROMELA code from the models of broadcast-based programs specified by other modelling languages.

Acknowledgment

This work is supported by NSFC grant 11671258, 11771280, National Science Foundation of Shanghai Municipal (17ZR1415400), National Natural Science Foundation of China under grant No.11361046, Ningxia High Education research fund with grant No.NGY2017180, undergraduate teaching project of Ningxia High Education with grant No.NXJG2016060, and the fund of Ningxia High Education Construction of First-Class Disciplines (Education) with grant No.NXYLXK2017B11, The Key Research and Development (Science and Technology Support Program) Program of Ningxia Province No.2018BEE03025, 2018BEE03026.

References

[1] D. Ferreira, V. Kostakos, and A. K. Dey, "Aware: mobile context instrumentation framework," *Frontiers in ICT*, vol. 2, p. 6, 2015.

[2] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 72–81.

[3] V. Vieira, K. Holl, and M. Hassel, "A context simulator as testing support for mobile apps," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 535–541.

[4] Z. Qin, Y. Tang, E. Novak, and Q. Li, "Mobiplay: A remote execution based record-and-replay tool for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 571–582.

[5] C. Luo, M. Kuutila, S. Klakegg, D. Ferreira, H. Flores, J. Goncalves, V. Kostakos, and M. Mäntylä, "How to validate mobile crowdsourcing design? leveraging data integration in prototype testing," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM, 2016, pp. 1448–1453.

[6] C. Luo, M. Kuutila, S. Klakegg, D. Ferreira, H. Flores, J. Goncalves, M. Mäntylä, and V. Skostakos, "Testaware: a laboratory-oriented testing tool for mobile context-aware applications," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 3, p. 80, 2017.

[7] H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying android applications using java pathfinder," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[8] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM computing surveys (CSUR)*, vol. 41, no. 4, p. 19, 2009.

[9] H. van der Merwe, O. Tkachuk, B. van der Merwe, and W. Visser, "Generation of library models for verification of android applications," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–5, 2015.

[10] G. J. Holzmann, "Design and validation of protocols: a tutorial," *Computer Networks and ISDN Systems*, vol. 25, no. 9, pp. 981–1017, 1993.

[11] ——, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[12] H. Flores, D. Ferreira, C. Luo, V. Kostakos, P. Hui, R. Sharma, S. Tarkoma, and Y. Li, "Social-aware device-to-device communication: a contribution for edge and fog computing?" in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM, 2016, pp. 1466–1471.

[13] H. Flores, P. Hui, S. Tarkoma, Y. Li, T. Anagnostopoulos, V. Kostakos, C. Luo, and X. Su, "Sensorclone: a framework for harnessing smart devices with virtual sensors," in *Proceedings of the 9th ACM Multimedia Systems Conference*. ACM, 2018, pp. 328–338.

[14] C. Luo, H. Koski, M. Korhonen, J. Goncalves, T. Anagnostopoulos, S. Konomi, S. Klakegg, and V. Kostakos, "Rapid clock synchronisation for ubiquitous sensing services involving multiple smartphones," in *Proceedings of the 2017 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2017 ACM International Symposium on Wearable Computers*. ACM, 2017, pp. 476–481.

[15] S. Klakegg, C. Luo, J. Goncalves, S. Hosio, and V. Kostakos, "Instrumenting smartphones with portable nirs," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM, 2016, pp. 618–623.

[16] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl, "A scalable and ontology-based p2p infrastructure for semantic web services," in *Peer-to-Peer Computing, 2002.(P2P 2002). Proceedings. Second International Conference on*. IEEE, 2002, pp. 104–111.

[17] R. Piyare, "Internet of things: ubiquitous home control and monitoring system using android based smart phone," *International Journal of Internet of Things*, vol. 2, no. 1, pp. 5–11, 2013.

[18] C. Luo, "Video summarization for object tracking in the internet of things," in *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on*. IEEE, 2014, pp. 288–293.