2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Systematic Testing of Reactive Software with Non-deterministic Events: A Case Study on LG Electric Oven

Yongbae Park, Shin Hong, Moonzoo Kim
CS Dept. KAIST, South Korea
Email: {yongbae.park, hongshin}@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

Dongju Lee, Junhee Cho
LG Electronics, South Korea
Email:{dongju81.lee, junhee.cho}@lge.com

***Abstract*—Most home appliance devices such as electric ovens are *reactive systems* which repeat receiving a user input/event through an event handler, updating their internal state based on the input, and generating outputs. A challenge to test a reactive program is to check if the program correctly reacts to various *non-deterministic* sequence of events because an unexpected sequence of events may make the system fail due to the race conditions between the main loop and asynchronous event handlers. Thus, it is important to systematically generate/test various sequences of events by controlling the order of events and relative timing of event occurrences with respect to the main loop execution. In this paper, we report our industrial experience to solve the aforementioned problem by developing a systematic event generation framework based on concolic testing technique. We have applied the framework to a LG electric oven and detected several critical bugs including one that makes the oven ignore user inputs due to the illegal state transition.**

I. Introduction

In the ubiquitous computing society, we utilize numerous devices controlled by software including complex ones such as PCs and smartphones to simpler ones such as electric ovens and refrigerators. Most such simple devices are *reactive systems* whose main operation is to repeat the following three tasks in a main event handling loop:

1) receiving an input through an input event handler (for example of an electric oven, the key event handler adds a key value of the auto-cook button to the input buffer when a user pushes the auto-cook button to start cooking food (see Figure 1 and Figure 2))
2) computing/updating internal state (e.g., an electric oven updates its internal state as a cooking mode and calculates electric voltage/current for the auto-cooking operation, which will be given to the heaters)
3) generating output (e.g., actuating the heaters and updating the LED display to show that the oven is cooking)

Note that an input event handler can receive an input event anytime. If an input event is given while the main loop is computing its internal state, the main loop is suspended and the event handler is executed instead; the main loop computation is resumed after the event handler completes its task.

However, this event-driven feature may cause race conditions between the event handlers and the main loop and result in *concurrency errors* unless proper synchronization between the main loop and the event handlers is enforced. In other words, because an input event can be given *non-deterministically*, the corresponding event handler may *unintentionally* interfere the main loop *anytime* by updating the global variables shared by the main loop. For example, suppose that an event handler adds a key input from a user into an input buffer and the main loop removes a key value from the buffer to react (see Figure 2). Without proper synchronization, the input event handler and the main loop can make the input buffer inconsistent and cause critical errors (see Section V-A).

Unfortunately, developers of home appliance devices often do not recognize this issue seriously because they think that home appliance devices such as electric ovens and refrigerators are simple enough to be free from complex concurrency problems. Consequently, such systems often suffer corner-case bugs that can be triggered by exceptional execution scenarios only. For example, if a user pushes a button and turn a dial at the same time, an electric oven may result in an abnormal state where it does not react to any button/dial and a user cannot control the oven at all (see Section V-B).

In this paper, we report our industrial experience to solve the aforementioned problem by developing a *systematic event generation framework*. The framework can systematically generate various sequences of events by controlling not only the order of events, but also relative timing of the event occurrences with respect to the main loop execution. The main idea of the framework is to utilize concolic execution (a.k.a. dynamic symbolic execution) to systematically generate events at every important execution point of the main loop (i.e., the framework can generate race conditions between the main loop and the event handlers if any. See Section III). Since the market for micro-controllers loaded on reactive systems is large (18 billion units in 2014 [17]) and will increase further with the advance of the IoT technologies, the benefit of the proposed technique for reactive systems will become more important.

We have applied the event generation framework to the controller software of a LG electric oven and detected several new bugs including atomicity violation bugs at the input buffer (Section V-A) and an illegal state transition bug (Section V-B) at system level, which make the oven fail to react to any button/dial and a user cannot control the oven at all. The contributions of this paper are as follows:

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.132

ICSE 2015, Florence, Italy
Software Engineering in Practice

- This paper addresses the challenges to test reactive systems with non-deterministic events in detail, particularly concurrency problems caused by race conditions between the main loop and the input event handlers. The clearly reported problem in the paper can help field engineers avoid possible threats in reactive software (Section V).
- We have developed a systematic and automated framework to generate test executions of various event sequences, which can improve the quality of the reactive software compared to the industrial practice of ad-hoc manual testing (Section III).
- We have demonstrated the effectiveness of the event generation framework to detect bugs in an industrial reactive system (i.e., a LG electric oven) by detecting new bugs (Section V).

The rest of this paper is organized as follows. Section II explains the overview of this testing project including the description of the LG electric oven. Section III explains the systematic event generation framework based on concolic testing technique. Section IV describes the testing setup of the project. Section V reports the testing results and Section VI discusses lessons learned from this project. Finally, Section VII summarizes the paper.

II. Project Overview

A. Background

Before starting this project in 2014, the authors at KAIST had collaborated with the LGE research department on automated testing techniques for embedded software using concolic testing (a.k.a., dynamic symbolic execution) [16] in 2013. Through the collaboration with KAIST, LGE was partly convinced of the advantages of concolic testing techniques in terms of the corner-case bug detection capability as shown in the several industrial applications [9], [8], [10], [11]. Thus, as the first step to adopt a new technology in a long term roadmap, LGE decided to start a project to apply concolic testing to simple home appliance products first.

This pilot project was five months long and the project team consisted of a professor and two graduate students from KAIST, a research engineer from the LGE research department, and a senior field engineer from the LGE production department for home appliance products. A main goal of this project was to develop a systematic testing framework based on concolic testing technique to detect corner-case bugs in the home appliance products of LGE. We target the controller software of an electric oven which has been on market for three years (Section II-B). The electric oven has a well-defined requirement specification for its behavior (see Figure 3). The full source code of the oven controller software was given to the KAIST authors. It took around two months for the KAIST authors to understand the domain knowledge of the electric oven and its controller code (Section II-C).

B. LG Electric Oven

Figure 1 shows a target LG electric oven. The target electric oven is a high-end multi-function electric oven and

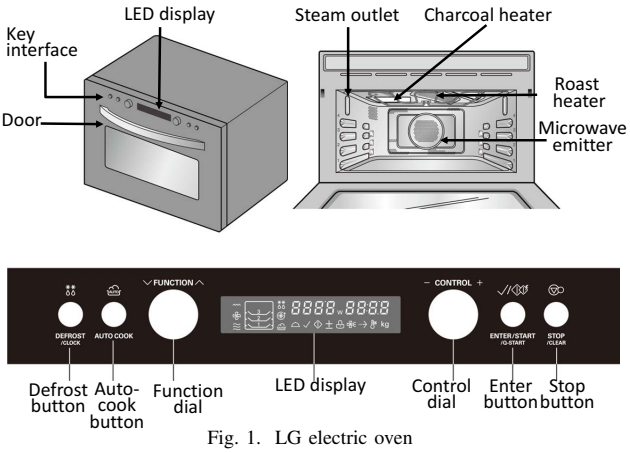

Fig. 1. LG electric oven

it provides dozens of pre-defined cooking recipes for dishes such as steaming dumpling and baking bread using the four heating mechanisms (steam, charcoal heater, roast heater, and microwave). The oven has temperature sensors, a door sensor, a cooling fan, and a lamp inside. The front panel of the oven has four buttons (defrost, auto-cook, enter, and stop buttons), two multi-function dials (a function dial and a control dial), and LED display at the center of the panel. The oven provides multiple functionalities which a user can select by a sequence of button and dial inputs (see Figure 3).

C. Electric Oven Controller Software

Figure 2 shows the architecture of the electric oven controller software. The controller software consists of a main loop, two input event handlers (a key/door event handler and a timed event handler), two output event handlers (a LED event handler and a cook command event handler), and the shared memory between the main loop and the event handlers.

- *Main loop:*
  For each iteration, the main loop updates the control state based on the given input data in the input buffer, and generates the hardware control commands such as LED command or cooking command. In addition, the main loop directly reads sensor data such as oven temperature.
- *Input event handlers:*
  The key/door event handler is invoked when a key event or a door event is given by a user (i.e., when a user presses a key or opens a door). The timed event handler is periodically invoked to report time progress (e.g., cooking time progress). These input event handlers transform an external input signal representing a physical event into input data and then store the data in the input buffer, which will be retrieved by the main loop.
- *Output event handlers:*
  The LED event handler and the cook command handler are periodically executed to convert the commands generated by the main loop to low-level signals which activate the oven hardware such as the LED display and a microwave emitter.

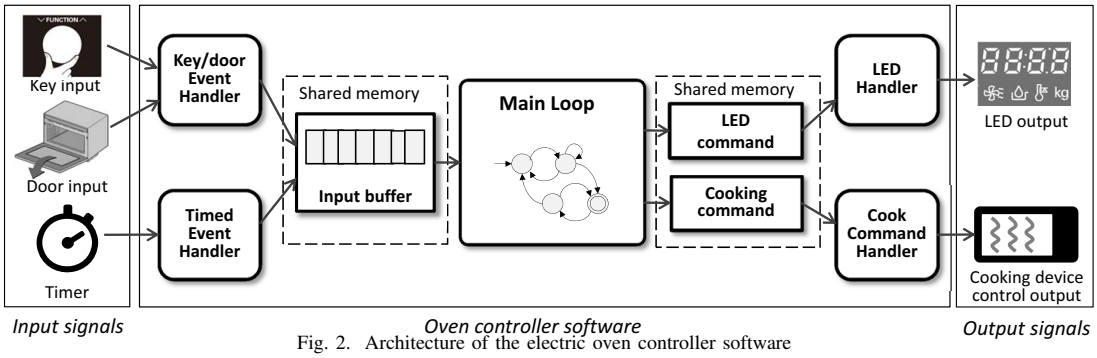

Fig. 2. Architecture of the electric oven controller software

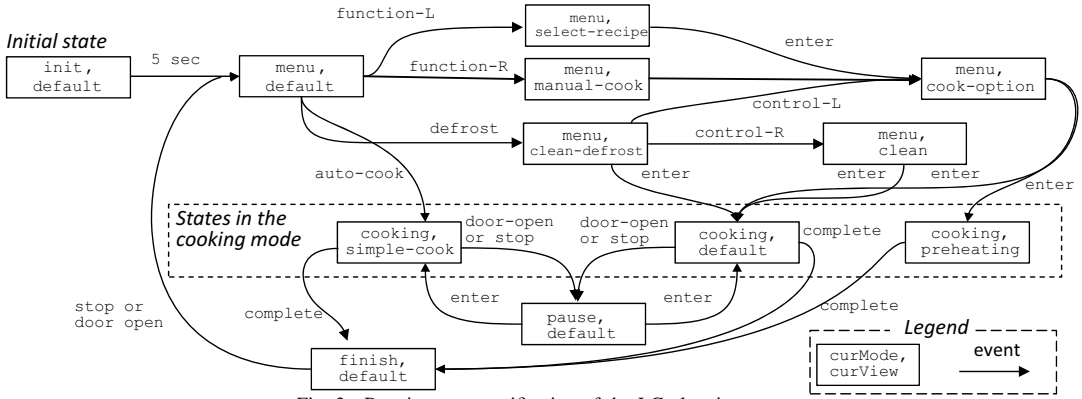

Fig. 3. Requirement specification of the LG electric oven

Figure 3 shows the (simplified) requirement specification on the controller software as an abstract state transition machine, which was specified by the original developers of the controller software. A state of this state machine consists of the two variables `curMode` and `curView` which represent a current operation mode and a current LED panel view of an oven respectively. `curMode` can be one of the following values:

- *init*: an initial mode when the oven boots up
- *menu*: a cook menu selection mode
- *cooking*: a mode where steam/heaters are being used
- *finish*: a mode representing the cooking is completed
- *pause*: a mode representing a situation where a user pauses cooking by pressing the stop button at *cook* mode.

`curView` can be one of *default*, *select-recipe*, *manual-cook*, *clean-defrost*, *cook-option*, *clean*, *simple-cook*, and *preheating* (each of which shows the LED display differently).

The state transition machine transits from one state to another based on a given input data (including key events, a door event, and time progress). The key events include events for the four buttons (i.e., *defrost*, *auto-cook*, *enter*, and *stop*) and events for the two dials (i.e., *function-L* and *function-R* that represents events of turning the function dial counter-clockwise or clockwise, respectively. Similarly, *control-L* and *control-R* are defined for the control dial). For example, when the oven is initially turned on, the state (i.e., `curMode` and `curView`) of the oven is set as (*init*,*default*) (see the left most state in the figure). After 5 seconds from the oven is turned on, the state changes to (*menu*,*default*) where a user can give a command through the four buttons and the two dials on the front panel. Note that the controller should *not* transit to "undefined" states that are not specified in the state transition machine. For example, when the oven state is in *cooking* mode (see the middle three states in the figure), its corresponding `curView` should be one of *simple-cook*, *default*, or *preheating*. [1]

The main loop can be preempted by an event handler but an event handler is non-preemptible (i.e., when an event handler runs, the main loop or another event handler cannot execute (no nested interrupt handlers allowed)). In other words, whenever an event is raised, CPU immediately suspends the main loop execution and starts executing a corresponding event handler; CPU resumes the main loop execution after the event handler completes its task.

The target control software has 658 functions in 180 files, and the total source code lines is 19,655 lines long (3505 branches) in C. The controller software implements an event by using an interrupt signal and an event handler as a function registered for the signal.

III. Systematic Event Generation Framework

We have developed an event generation framework that can systematically control event generation in the following two ways (and their combination) where e_1, e_2, and e_3 indicate different events:

- *Controlling the order of events:*
  The framework can generate multiple sequences of events such as $e_1.e_2.e_3$..., $e_1.e_3.e_2$..., $e_2.e_1.e_3$..., and so on.

[1]The full state machine specification defines a state as a tuple of the five variables and contains 122 states and 634 transitions. Figure 3 is an abstract version and has non-deterministic transitions due to the abstraction.

- *Controlling the relative timing of the event occurrence with respect to the main loop execution:*
  The framework can generate multiple sequence of events such as $[e_1@l_1.e_2@l_2.e_3@l_3]_1[]_2$..., $[e_1@l_1.e_2@l_3]_1[e_3@l_5]_2$..., $[]_1[e_1@l_3.e_2@l_3.e_3@l_4]_2$... where l_1, l_2, ... are the code locations in the main loop and $[e_3@l_5]_2$ means that e_3 occurs right before the main loop executes the statement at l_5 at its second iteration. Note that for the same order of events $e_1.e_2.e_3$, there exist various sequences of different relative timing cases.

The framework can instrument the main loop of the target C source code by statically inserting probes at every important code location in the main loop (Section III-A). Then, the probes invoke event handlers systematically by controlling each probe through concolic testing (Section III-B). Since an event generation immediately leads to invocation of an event handler that is registered to handle the event, the inserted probes directly invoke event handlers (without generating events) at "important" execution point of the main loop. "important" code location varies depending on the instrumentation strategy (see Figure 4).

A. Instrumentation of Target Program

First, a user should specify a segment of the target program as the main loop by adding `#pragma SEGF start` and `#pragma SEGF end` before and after the segment. The framework inserts the probes in the specified main loop segment recursively so that the bodies of the callee functions in the segment will be instrumented with the probes and so on (see `main_task()` in Figure 4(b)).

We have developed the following two strategies to insert the probes, which have different bug detection capabilities and different runtime costs:

- *Statement-based strategy* inserts a probe at every source code statement in the specified target code segment.
- *Basic block-based strategy* inserts the probe at the beginning of every basic block in the target code segment.

The statement-based strategy inserts more probes than the basic block-based one because a basic block usually contains multiple statements. Consequently, the statement-based strategy will generate execution scenarios where event handlers are invoked more frequently than the execution scenarios generated by the basic block-based strategy. Thus, we can expect that the statement-based strategy will have higher bug detection capability but require more testing time than the basic block-based strategy.

Figure 4(a) shows example target code to insert the probes. Suppose that a user specifies line 5 as the main loop by inserting line 4 (`#pragma SEGF start`) and line 6 (`#pragma SEGF end`). Also, suppose that `ev1Hdl()` at lines 14-15 is registered as an event handler for events of the `ev1` type (for example, the electric oven has 10 events of the key/door type including *enter*, *stop*, etc. (see Section II-C)). Figure 4(b) shows the instrumented target code by the statement-based strategy. The framework inserts a probe `p(1)` before `main_task()` at line 5 where `1` is a probe ID which is a unique number for each probe. Also, the framework inserts probes into the body of `main_task()` (i.e., inserting `p(2)`, `p(3)`, and `p(4)` at lines 10 to 12). Similarly, the framework inserts probes into the body of `f()` recursively (not shown in the figure). Note that the framework does not insert a probe at lines 14-15 because `ev1Hdl()` is not called from `main_task()` or its sub-functions. Figure 4(c) shows the instrumented target code by the basic block-based strategy. The framework inserts probes at only line 5 and line 10 because Figure 4(c) has only two basic blocks beginning at line 5 and line 10, respectively.

Two graduate students spent 2 months to develop the idea of the current systematic event generation mechanism and implement the framework. The event generation framework is written in 1540 lines of C++ code using Clang library.

B. Systematic Event Generation

The event generation framework systematically generates various sequences of event handler invocations including various relative timing of event handler invocations with respect to the main loop execution through concolic execution.

Figure 5 shows the pseudo code of `probe_ev1()` that can invoke `ev1Hdl()` which is registered to handle an event of the type `ev1`. `ev1Loc` at line 1 is a two dimensional symbolic array whose values decide which probe will invoke `ev1Hdl()` at a specified main loop iteration. `ev1Loc[NUM_ITER_EV1][MAX_EV1_OCCUR]` at line 1 indicates that the instrumented target program will invoke `ev1Hdl()` at most `MAX_EV1_OCCUR` times at each iteration of the main loop for total `NUM_ITER_EV1` iterations. `ev1Loc[`i`]` contains a sub-array that has a list of the IDs of the probes each of which will invoke `ev1Hdl()` once at the $i+1$ th iteration (i.e., `ev1Loc[`i`][`j`]` indicates an ID of a probe that will invoke `ev1Hdl()` at the $i+1$ th iteration). For example, `ev1Loc[1][0]=3`, `ev1Loc[1][1]=0`, `ev1Loc[1][2]=2` indicates that the `ev1Hdl()` will be invoked by the probes whose IDs are 3 and 2 at the second main loop iteration. Note that `ev1Loc[1][1]` is ignored because 0 is not a valid probe ID (i.e., no probe has an ID 0). `iter` at line 2 indicates the current iteration of the main loop (`iter` will be increased by one at the end of every main loop iteration). Lines 4 to 7 declare each element of `ev1Loc` as a symbolic integer variable. We use CREST-BV [10] to perform the dynamic symbolic execution, which is an instrumentation based dynamic symbolic execution tool for C programs (faster than KLEE [10]) (CREST-BV is the extension of CREST [1] by supporting bit-vector arithmetic).

Lines 9 to 18 explains `probe_ev1().isInHdl` at line 10 indicates if a current probe is being executed by an event handler. A probe should not invoke an event handler if it is called by an event handler since the oven software does not allow nested interrupt handlers (Section II-C). [2] If `isInHdl` is false

[2]A probe may be executed by an event handler because some functions may be called by both main loop and the event handler.

```
 1 …
 2 int main() {
 3   while (…) {
 4   #pragma SEGF start
 5   main_task();
 6   #pragma SEGF end}}
 7
 8 void main_task() {
 9   int x;
10   x=10;//a local var
11   f(y);//a global var
12   f(*ptr);}
13
14 void ev1Hdl(){
15  … ; y++; …}
```

(a) Original target source code

```
 1 …
 2 int main() {
 3   while(…) {
 4   #pragma SEGF start
 5   p(1); main_task();
 6   #pragma SEGF end}}
 7
 8 void main_task() {
 9   int x;
10   p(2); x=10;
11   p(3); f(y);
12   p(4); f(*ptr);}
13
14 void ev1Hdl(){
15  … ; y++; …}
```

(b) Instrumented source code by the statement-based strategy

```
 1 …
 2 int main() {
 3   while(…) {
 4   #pragma SEGF start
 5   p(1); main_task();
 6   #pragma SEGF end}}
 7
 8 void main_task() {
 9   int x;
10   p(2); x=10;
11   f(y);
12   f(*ptr);}
13
14 void ev1Hdl(){
15  … ; y++; …}
```

(c) Instrumented source code by the basic block-based strategy

Fig. 4. Example showing how the two strategies insert the probes

```
01:int ev1Loc[NUM_ITER_EV1][MAX_EV1_OCCUR];
02:int iter=0;
03:
04:void init(){
05:  for(i=0;i<MAX_ITER_EV1;i++)
06:    for(j=0;j<MAX_EV1_OCCUR;j++)
07:      sym_int(ev1Loc[i][j]);}
08:
09:void probe_ev1(int probeId){
10:  static int isInHdl = FALSE;
11:  if(!isInHdl){
12:    for(j=0;j<MAX_EV1_OCCUR;j++){
13:      if(ev1Loc[iter][j]==probeId){
14:        isInHdl = TRUE;
15:        ev1Hdl();
16:        ev1Loc[iter][j]=COMPLETED;
17:        isInHdl = FALSE;
18:} } } }
```

Fig. 5. Pseudo code of a probe for the event type `ev1`

and the current probe is the one to invoke `ev1Hdl()` (i.e., `ev1Loc[iter][j] == probeId` at line 13), `isInHdl` is set as true and `ev1Hdl()` is invoked. After `ev1Hdl()` completes its task, the current element of `ev1Loc` is marked as completed (line 16) and `isInHdl` is set back to false (line 17).

For example, a sequence of events $[e@l_1.e@l_3]_1[e@l_5]_2$ can be generated by setting `evLoc[0]`=$\{1,3,0\}$ and `evLoc[1]`=$\{5,0,0\}$ with the assumption that each main loop iteration generates maximum three events. In this way, the framework can generate various execution scenarios by systematically invoking event handlers at every important execution points of the main loop.

Finally, the event generation framework performs the aforementioned task for every event type separately at each inserted probe. Thus, the framework can comprehensively generate various sequences of invocations of event handlers including exceptional ones such as an execution that contains multiple invocations of an event handler at the single code location of the main loop (see Figure 10).

C. Related Techniques

Researchers have applied model checking techniques such as Verisoft [4] and SPIN [6] to find bugs in reactive programs. For example, SPIN was used to verify an event-driven network server product of Lucent Technology [7] and Verisoft was used to test an CDMA library of Lucent Technology [2]. A limitation of these techniques for industrial application is, however, that a user has to write an abstract model of the target program (or specify a non-deterministic execution environment by using `VS_toss(n)` for Verisoft), which is not affordable for most industrial software projects under hard time-to-market pressure. Compared to these model checking techniques, our approach is more affordable to industrial setting because our framework, with relatively less human effort, can generate test executions with various sequences of events including relative timing of the events in fine granularity.

The idea of testing a sequential version of a concurrent program has been investigated in the bounded model checking of multithreaded programs [14]. Still, there are few techniques that support event-driven reactive programs. Regehr et al. [15] presents a technique that inserts random interrupt-raising probes to the target software (TinyOS applications) to generate various interrupt execution scenarios. Kotker et al. [12] presents a testing technique that utilizes sequential versions of interrupt-driven programs for timing analyses. Unlike these work, our technique *systematically* generates various event scenarios and also various input values by concolic testing technique to detect functional errors in event-driven reactive software.

IV. Testing Oven Controller Software with the Event Generation Framework

We first applied the framework to test units of the controller (Section IV-A) and then to test the entire controller software (Section IV-B). Also, we applied noise-based random testing techniques for comparison (Section IV-C). The experiments were performed on the machine with Intel I5 3.6 Ghz and 16 Gigabyte memory, which runs 64 bits Ubuntu linux.

A. Unit-level Testing

We have applied the framework to the following three units:

- a circular queue (calling it CQ) (145 lines of C code with 7 functions) to which the input event handler store input

values and from which the main loop loads input values (*input buffer* in Figure 2).
- a load module that enforces the voltage to the heaters
- a model option unit that recognizes the oven hardware and enables/disables relevant oven features.

We selected these three units to apply the event generation framework because the original developer of the oven controller put high priority to test these three units due to the significance of these three units for reliable oven products. In particular, the correctness of a circular queue (CQ) is very crucial because the main loop computes and updates the state of the oven based on the data obtained from CQ. In addition, CQ is a general unit which can be reused in other products of LGE. Thus, it is important to detect bugs in CQ if any. We focus to describe how we applied the event generation framework to test CQ in detail. [3]

1) CQ Data Structure: CQ contains the following variables:
- `qArray` is an array that serves as a buffer for input values
- `headIdx` points to an element of `qArray` to pop up
- `tailIdx` points to an element of `qArray` to store a new input value
- `queueFull` indicates if the queue is full. If `headIdx==tailIdx` and `queueFull` is false, the queue is empty. If `headIdx==tailIdx` and `queueFull` is true, the queue is full.

In addition, CQ uses `dequeue()` (see Figure 8) and `enqueue()` to add (store) and remove (pop up) a new value to/from the queue.

2) Unit Testing Setup for CQ : Figure 6 explains how we setup the unit testing driver for CQ. `test_init()` at line 3 initialize the data structure of CQ symbolically as follows, to generate various execution scenarios:
- `qArray`: We set the size of qArray as *three* which is a minimal number to represent various situations such as `headIdx` $\neq$ `tailIdx` and `qArray` has a valid element which is pointed by neither of these two variables (see Figure 10). In addition, `qArray` is initialized to have three concrete values 1, 2, and 3.
- `headIdx` and `tailIdx` are declared as symbolic integer variables whose ranges are between zero and two because the size of `qArray` is three.
- `queueFull` is declared as a symbolic Boolean variable such that if `queueFull==true`, `headIdx` must be equal to `tailIdx`.

As a result, `test_init()` represents all possible states of CQ whose size is three.

We specified line 7 of Figure 6 as the main loop body which removes a value from CQ. Also we set up that the input event handler calls `enqueue(v++)` where `v` is initially 4. We tried to setup symbolic environment minimal to represent various scenarios but still avoid unnecessarily large symbolic search space that will increase the testing time.

[3]To secure the intellectual property rights of LGE, information on the units is not written in the paper except CQ that uses a publicly available algorithm.

```
 1:...
 2:int main() {
 3:  test_init();
 4:  int qSize = getQSize();
 5:  for (i=0; i < qSize; i++) {
 6:    #pragma SEGF start
 7:    data=dequeue();
 8:    #pragma SEGF end
 9:    readData[count++] = *data; }
10:
11:  for(i=0; i < count; i++)
12:    assert(readData[i]==writtenData[i]);
13:}
```

Fig. 6. Unit testing driver for CQ

We configured the event generation framework to generate executions that run the main loop three times and invoke the input event handler at most twice per main loop iteration (i.e., `NUM_ITER_EVKEY`=3 and `MAX_EVKEY_OCCUR`=2).

As a test oracle, lines 11 and 12 check if the values read by `dequeue()`s are equal to the values written by `enqueue()`s.

We used two search strategies for dynamic symbolic execution: DFS and random negation which randomly selects a branch condition to negate. We tested the instrumented CQ for 30 minutes per search strategy. For the random negation search strategy, we repeated the testing 30 times.

B. Integration Testing

We have applied the event generation framework to the integrated controller software after removing the functions that have heavy hardware dependency such as a EEPROM module and a heater driver module. Note that the event generation framework serves as a hardware emulator to invoke event handlers for physical events so that we can test most part of the controller software without the real hardware. The target controller software we tested contains 527 functions in 12,691 lines of C code which is around 80% of the all functions or 65% of the all lines of the controller software.

The target controller program has one main loop with the four event handlers (i.e., the key/door event handler, the timed event handler, the LED event handler, and the cook command handler in Figure 2). We built a symbolic environment to invoke the input event handlers with various key/door events (e.g., $enter$, $stop$, $function\text{-}L$, $door\text{-}open$, etc) and timed events systematically. For the main loop, we modified the main loop code to iterate twelve times (i.e., `NUM_ITER_EV` = 12). We configured the event generation framework to invoke an event handler at most n times for each of the four event types (i.e., a key/door event type, a timed event type, a LED event type, and a cook command event type) per main loop iteration where $n \in \{2, 3, 4\}$ (i.e., `MAX_EV_OCCUR` $\in \{2, 3, 4\}$).

For test oracles, the full state machine specification is utilized (see Figure 3 which is the abstract version of the full state transition machine). In other words, `assert()` statements are inserted at the statements that make a state transition to check if the current state transition is valid (i.e., following the state machine specification).

We used two search strategies for dynamic symbolic execution: DFS and random negation. We tested the instrumented

```
01:int main() {// a main loop thread
02:  ...
03:  while(...) {
04:    delay(200)); main_task();
05:  }}
06:
07:void evHnd() { ... }
08:
09:void *evGen(...){//an event gen thread
10:  while(...) {
11:    usleep(rand()% MAX_EVENT_GEN_SLEEP);
12:    pthread_kill(main_thread...); }}
```

Fig. 7. Noise injection based random testing

the controller software for 1 hour per search strategy. For the random negation search strategy, we repeated the testing 30 times.

C. Noise Injection based Random Testing technique

To demonstrate the effectiveness and the efficiency of the event generation framework through comparison, we applied noise injection based random testing techniques to the target program. A noise injection based random testing is a popular technique to test concurrent programs because it can detect concurrency bugs without complex analysis of the target program [3], [13].

Figure 7 shows the pseudo code of the noise injection based random testing framework. To apply a noise injection based random testing, we create 2 threads; one thread runs the main loop and the other thread repeatedly generates events (i.e., generates signals to the main thread by invoking `pthread_kill(main_thread, SIGUSR<i>)`). When the main thread receives an event (i.e., an interrupt signal `SIGUSR<i>`), the main thread suspends its current execution and executes the registered event handler (e.g., `evHnd()` at line 7).

To inject random timing noise, we insert timing delays to both the main loop thread and the event generation thread to diversify event generation scenarios.

- *Main loop thread:*
  As shown at line 4, we insert 200 microseconds delay at every statement in the main loop and its callee functions recursively (in the similar way of the statement-based strategy to insert the probes) so that an event generation thread can probabilistically raise multiple events during the 200 microseconds delay at each statement of the main loop. [4]
- *Event generation thread:*
  We insert the following three *maximum* timing delays at each event generation: 500, 1000, and 1,500 microseconds (i.e., `MAX_EVENT_GEN_SLEEP` at line 11 can be 500, 1000 or 1500).

For the random unit testing of CQ, we used the similar unit testing driver in Figure 6 except that CQ is initialized randomly not symbolically. We tested CQ for 30 minutes and repeated the random testing 30 times.

[4] An event generation thread takes around 65 microseconds on average to execute `usleep()` and `pthread_kill()` (lines 11-12 of Figure 7) because `usleep(0)` takes 60 microseconds on average due to system call overhead. Thus, the 200 microseconds delay can allow generating three events at each statement of the main loop.

```
1:void* dequeue() {
2:  void* result = NULL;
3:  if (!isEmpty()) {
4:    result = headIdx;
5:    headIdx = getNextIdx(headIdx);
      /* the inconsistency error can occur
         if two enqueue()s occur here. */
6:    queueFull = false;
      /* the overwriting error can occur
         if enqueue() occurs here. */
7:  } else  result = NULL;
8:  return result; }
```

Fig. 8. Buggy `dequeue()` of the circular queue CQ

For the integration testing, the random testing uses the similar integration testing driver used for the event generation framework except that input events to generate are selected randomly. We tested the controller software for 1 hour and repeated the random testing 30 times.

V. Testing Results on LG Electric Ovens

This section describes the results of applying the systematic event generation framework to the LG electric oven. We spent a month to apply the framework to the controller.

A. Result of the Unit Testing

We detected the following two atomicity violation bugs in CQ (Section IV-A) by using the event generation framework [5].

- an *overwriting bug* which causes the queue to overwrite the oldest value in the queue with a new value and cause `dequeue()` (see Figure 8) to incorrectly return the new value instead of the oldest value
- an *inconsistency bug* which causes the queue to lose all values in the queue because the queue considers itself empty while it is not

Figure 8 describes a simplified `dequeue()` of the circular queue which has these two bugs.

Figure 9 illustrates the overwriting bug. Suppose that the size of the queue is 3 and the queue contains three elements 1, 2, and 3 as specified in the testing setup of CQ (Section IV-A). After executing line 4 of `dequeue()`, `result` points to the first element of `qArray`. After executing line 5 and line 6, `headIdx` points to the second element and `queueFull` becomes false, respectively. Suppose that `enqueue(4)` is invoked between line 6 and line 8 (`enqueue()` can proceed only when `queueFull` is false). Note that `enqueue(4)` overwrites the first element with 4 because `tailIdx` points to the first element and `queueFull` is false. As a result, `dequeue()` will return 4 instead of the oldest value 1. A main cause for the original developers to miss this bug is that they could not imagine or test an execution scenario where

[5] The overwriting bug and the inconsistency bug are *multi-variable* atomicity violation bugs [5] on `headIdx`/`queueFull` and `queueFull`/`*result`, respectively. However, we decide to use a term 'atomicity bug' in this paper because field engineers are more familiar with the term.

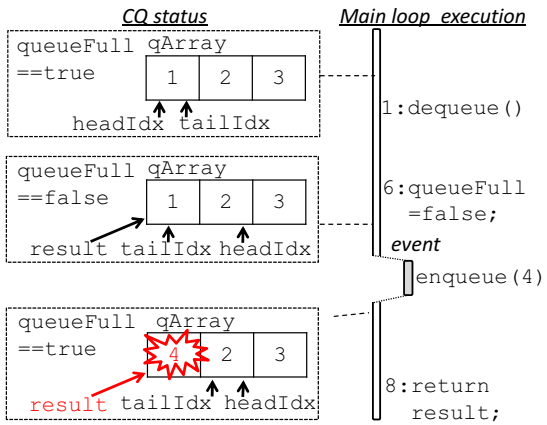

Fig. 9. Error caused by the overwriting bug

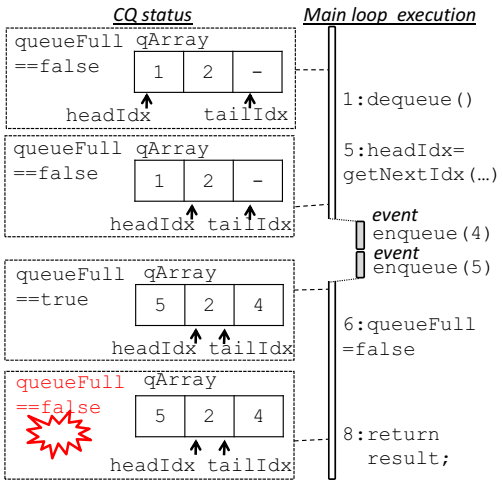

Fig. 10. Error caused by the inconsistency bug

the input event handler which calls `enqueue()` is invoked between line 6 and line 8 of `dequeue()` when the queue is full.

Figure 10 illustrates the inconsistency bug. Suppose that the size of the queue is 3 and the queue contains only two valid elements 1 and 2. After executing line 5 of `dequeue()`, `headIdx` points to the second element. Suppose that `enqueue(4)` and `enqueue(5)` are executed by the event handler between line 5 and line 6. Then, the third element and the first element have 4 and 5, respectively. After executing line 6, although the queue contains 5, 2, and 4, the queue considers itself empty because `queueFull` becomes false. We fixed these two bugs by modifying the circular queue algorithm and related variables.

Note that it is more difficult to detect the inconsistency bug than the overwriting bug because the bug triggering condition for the inconsistency bug (i.e., *two consecutive* invocation of the input event handler between line 5 and line 6) is stronger than the condition for the overwriting bug (i.e., one invocation of the input event handler between line 6 and line 8).

Table I shows the testing results of the event generation framework and the random techniques. The systematic event generation framework inserted 17 and 11 probes in CQ by the statement-based and basic block-based, strategies, respectively. For example, the statement-based strategy detects the inconsistency bug in 15.30 seconds (after executing CQ 1128 times) with the DFS search strategy (see the second column of the third row in the table). With the random search strategy, the statement-based strategy detects the bug at every testing run (i.e., 30 minutes testing) of the 30 testing runs; it detects the bug in 339.63 seconds on average after executing CQ 19418.60 times. But the basic block-based strategy failed to detect the bug (indicated as 'N/A' in the table) because it did not insert the probe between line 5 and line 6 of Figure 8 that are in the same basic block.

Compared to the event generation framework, the random testing technique completely failed to detect the inconsistency bug with maximum 1,500 microseconds delay at event generation (see the third row and the 10th column of the table). Although the random testing technique detected the bug with maximum 500 and 1,000 microseconds delays, the probability to detect the bug is only 20% and 23%, respectively (see the sixth and eighth columns of the third row of the table). Consequently, the average time taken to detect the bug for these two delays is more than 100 (=1577.90/15.30) times longer than the event generation framework.

This is because the bug triggering scenario for the inconsistency bug is a very exceptional one and the probability for the random technique to synthesize this scenario is very low (also note that random techniques may generate redundant execution scenarios repeatedly which wastes testing time). In contrast, the event generation framework systematically tries to analyze all execution scenarios with the DFS search strategy, which can certainly detect the bug much faster than the random technique. For the overwriting bug which is easier to detect than the inconsistency bug, random testing detected the bug faster than the event generation framework. Both the event generation framework and the random testing techniques cover around 70% of the branches of CQ.

B. Results of the Integration Testing

Through the integration testing, we observed more than 100 assert violations. For example, we found that the controller made an illegal state transition from the state (*menu*,*default*) to (*cooking*,*select-recipe*), which is an *undefined* state. In other words, the full state machine specification has no such state. Thus, once the oven controller gets into the undefined state, the oven fails to react to any user input.

This illegal transition was made by the two consecutive events *auto-cook* and *function-L* at the *same* main loop iteration on $(menu,default)$ state. In other words, the error occurs when a user presses the auto-cook button then *immediately* turns the function dial counter-clockwise when the oven is in the menu mode. This error does not occur if a user presses

TABLE I
TIME TO DETECT THE BUGS IN CIRCULAR QUEUE

	Event generation framework				Random testing					
	Statement		Basic Block		0–500 μsec.		0–1000μsec.		0–1500 μsec.	
	DFS	Random	DFS	Random	Detect. rate	Detect. time	Detect. rate	Detect. time	Detect. rate	Detect. time
Overwriting bug	3.00 (228)	0.47 (35.13)	N/A	N/A	1.00	0.01 (1.27)	1.00	0.01 (1.97)	1.00	0.01 (2.57)
Inconsistency bug	15.30 (1128)	339.63 (19.41K)	N/A	N/A	0.20	1653.00 (0.33M)	0.23	1577.90 (0.31M)	0.00	1800.00 (0.36M)

TABLE II
TIME TO DETECT THE BUG IN THE CONTROLLER PROGRAM

Techniques			Detect. time	# of exec.
Event generation framework	$n = 2$	STMT	195.10	3991.40
		BB	172.63	3800.33
	$n = 3$	STMT	208.60	2920.60
		BB	249.43	3845.10
	$n = 4$	STMT	280.03	3210.60
		BB	249.63	3307.43
Random testing		500 μsec.	22.63	1.77
		1000 μsec.	23.63	1.97
		1500 μsec.	24.30	1.47

the auto-cook button and then turns the function dial not immediately (e.g., with 0.5 second interval between the two actions). The original oven developers confirmed this problem by replaying the erroneous scenario with the real oven device.

After analyzing the erroneous test executions, we found a bug at the function $f_{mt}()$ which makes *multiple state transitions* in one main loop iteration. Using $f_{mt}()$, the controller can handle multiple events fast in one main loop iteration. But this makes the controller program complicated and $f_{mt}()$ does not operate correctly with unexpected event sequences.

In the above erroneous execution, $f_{mt}()$ updates the current state from (*menu*,*default*) to (*cooking*,*simple-cook*) with *auto-cook* event first (see Figure 3). Then, with *function-L* event, $f_{mt}()$ incorrectly updates the current state based on the previous state (i.e., (*menu*,*default*)) not the recently updated state (i.e., (*cooking*,*simple-cook*). Since some event such as *function-L* may update the current state partially (i.e., updating only `curView`, not `curMode`), $f_{mt}()$ updates `curView` to *select-recipe* with *function-L* (as shown at the top of Figure 3) because $f_{mt}()$ think that the current state is still (*menu*,*default*). As a result, $f_{mt}()$ updates the current state as (*cooking*,*select-recipe*) which is an undefined state.

A main cause for the original developers to miss this bug is that they could not imagine or test an execution scenario where a user presses the auto-cook button and turns the function dial almost same time (i.e., at the same main loop iteration). After fixing $f_{mt}()$, all assert violations were removed.

Table II shows the testing results of the event generation framework and the random techniques on the controller software. The systematic event generation framework inserted 5009 and 2339 probes by the statement-based and basic block-based strategies, respectively. The event generation framework detected the illegal transition bug with the random search strategy, but not with DFS at all. With the random search strategy, the basic block-based strategy (BB) with `MAX_EV_OCCUR`=2 detects the bug at every testing run (i.e., 1 hour testing) of the 30 testing runs. It detects the bug in 172.63 seconds after executing the controller software 3800.33 times on average (see the second row of the table). In addition, the bug detection time increases as `MAX_EV_OCCUR` increases from 2 to 4 (i.e., from 172.63 seconds to 249.63 seconds) because larger `MAX_EV_OCCUR` makes larger search space, which requires more time to detect the bug. The event generation framework and the random testing cover around 55% of the branches of the target controller software.

The random testing techniques detected the bug 7.6 times faster than the event generation framework with the basic block-based strategy (BB) with `MAX_EV_OCCUR`=2. For example, with the maximum timing delay of 500 microseconds at the event generation thread, the random testing technique detected the bug in 22.63 seconds after executing the program 1.77 times on average (see the eighth row of the table). Note that the symbolic search space of this integration testing is large (i.e., by containing more than 100 symbolic variables and each execution generates around 6,000 symbolic conditions to solve on average due to the large number of the inserted probes). Therefore, the event generation framework based on the concolic testing was slower than the random testing to detect the illegal state transition bug in the control software.

VI. LESSONS LEARNED

A. Effectiveness of the Event Generation Framework

Through the project, we confirmed that the event generation framework can detect critical corner-case bugs effectively (Section V). This is because the framework can generate various timing scenarios of the event occurrences systematically based on concolic testing including exceptional ones which human engineers cannot think (Section III). Thus, by applying the framework, developers can effectively improve the quality of industrial reactive software.

B. Systematic Testing vs. Random Testing

We have compared the event generation framework with the carefully designed random testing techniques (Section IV-C). We observed that the systematic framework detected the corner-case bug (i.e., the inconsistency bug) 100 times faster than the random testing on small unit (i.e., CQ) because the probability for the random techniques to synthesize the corner-case execution scenarios is very low due to the generation of the redundant test executions. However, we observed that the

huge symbolic search space is a bottleneck for the framework; the random testing techniques were 5 times faster than the framework on the whole controller software whose testing generates huge symbolic search space (Section V-B).

Thus, it is beneficial to utilize various automated testing techniques together because they have different characteristics. For example, a user can apply the event generation framework to unit testing first but apply the carefully designed random techniques to system level testing first.

C. Industrial Adoption of the Advanced Testing Techniques

Through the discussion with the LGE field engineers, we could make the following observations for the successful technology transfer.

1) High Demand of Corner-case Bug Detection for Home Appliance Domain: In general, home appliance developers are sensitive to corner-case bugs because home appliances can make tragic physical accidents (e.g., an electric oven may explode). Also, the relatively long lifetime of the home appliance products encourages developers to improve the quality of their products. Thus, the original developers of the electric oven appreciated the bug reports and showed high interest to the framework. As a result, LGE and KAIST plan to improve the event generation framework and apply the framework to three more home appliance domains in 2015.

2) Necessity of Training Developers: Another reason for the smooth acceptance of the framework by the developers is that the developers were already exposed to advanced software analysis techniques before the project began. For example, one of the developers worked on model checking during his master study. Also, one KAIST author made a series of the eight lectures on dynamic symbolic execution including detailed tool design of CREST to LGE developers in 2012. Thus, the developers can estimate the benefit and the manual effort required to apply the new technique to their products and feel more comfortable to adopt the technique.

D. Technical Challenges

Through the project, we identified the following technical challenge to improve the quality of target software further.

1) Outdated Requirement Specification: We confirmed the importance of the requirement specification by utilizing the state machine specification to detect the illegal state transition bug (Section V-B). However, we had to revise the specification with the help of the original developers since the original specification was outdated. It might be necessary to develop a technique to generate static/dynamic invariant constraints and utilize the constraints as test oracles since test oracle generation is important for testing but still largely dependent on human engineers.

2) Micro-controller Specific Low-level Compilation: Most home appliance software are compiled using the micro-controller specific compilers, which sometimes compile source code in a non-standard way due to the hardware characteristics. For example, an original developer told us that, for hardware dependent variables, sometimes integer type casting does not follow the standard C semantics. We could not find problems caused by such issues because we did not test the hardware dependent functions in this project. We will try to analyze such low-level issues in the next year project.

VII. Conclusion and Future Work

We reported our industrial experience to test a real-world reactive software with non-deterministic events using the systematic event generation framework based on concolic testing technique. We detected several critical errors in the controller software, which had not been detected by the field engineers before. As a result, through the application of the proposed systematic event generation framework, we could improve the quality of the LG electric ovens in practice. This project result was evaluated high by LGE and we plan to apply the framework to three more target domain next year and extend the framework to resolve the technical issues found in the project.

Acknowledgment

This work is supported in part by the NRF Mid-career Research Program funded by the MSIP, Korea (NRF-2012R1A2A2A01046172), the ITRC support program funded by the MSIP, and supervised by the NIPA, Korea (NIPA-2014-H0301-14-1023), and CTO Software Center in LG Electronics.

References

[1] J. Burnim. CREST - automatic test generation tool for C. http://code.google.com/p/crest/.
[2] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: An industrial case study. In *ICSE*, 2002.
[3] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded Java program test generation. In *ACM-ISCOPE Conference on Java Grande (JGI)*, 2001.
[4] P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *CAV*, 1997.
[5] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set serializability violations. In *ICSE*, 2008.
[6] G. Holzmann. *The Spin Model Checker*. Wiley, New York, 2003.
[7] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *ICSE*, 1999.
[8] M. Kim, Y. Kim, and Y. Choi. Concolic testing of the multi-sector read operation for flash storage platform software. *Formal Aspects of Computing (FAC)*, 24(2), 2012.
[9] M. Kim, Y. Kim, and Y. Jang. Industrial application of concolic testing on embedded software: Case studies. In *ICST*, 2012.
[10] Y. Kim, M. Kim, Y. Kim, and Y. Jang. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In *ICSE SEiP track*, 2012.
[11] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim. Automated unit testing of large industrial embedded software using concolic testing. In *ASE experience track*, 2013.
[12] J. Kotker, D. Sadigh, and S. A. Seshia. Timing analysis of interrupt-driven programs under context-bounds. In *FMCAD*, 2011.
[13] B. Křena, Z. Letko, T. Vojnar, and S. Ur. A platform for search-based testing of concurrent software. In *PADTAD*, 2010.
[14] S. Qadeer and D. Wu. KISS: Keep It Simple and Sequential. In *PLDI*, 2004.
[15] J. Regehr. Random testing of interrupt-driven software. In *EMSOFT*, 2005.
[16] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.
[17] Electronics Purchasing Strategies. Microcontroller market rebounds in 2014, August 2014. http://electronicspurchasingstrategies.com/2014/08/14/microcontroller-market-rebounds-2014/.