석사 학위논문
Master's Thesis

비결정적 이벤트를 처리하는 반응형 소프트웨어를 위한 자동화 테스팅 기법: LG 전기 오븐 사례 연구

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

박 용 배 (朴 容 培 Park, Yongbae)
전산학과
Department of Computer Science

KAIST

2015

비결정적 이벤트를 처리하는 반응형 소프트웨어를 위한 자동화 테스팅 기법: LG 전기 오븐 사례 연구

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

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

Advisor : Professor Kim, Moonzoo

by

Park, Yongbae

Department of Computer Science

KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Department of Computer Science . The study was conducted in accordance with Code of Research Ethics[1].

2014. 12. 16.

Approved by

Professor Kim, Moonzoo

[Advisor]

[1]Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

비결정적 이벤트를 처리하는 반응형 소프트웨어를 위한 자동화 테스팅 기법: LG 전기 오븐 사례 연구

박 용 배

위 논문은 한국과학기술원 석사학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2014년 12월 16일

심사위원장 김 문 주 (인)

심사위원 송 준 화 (인)

심사위원 한 동 수 (인)

MCS
20133276

박 용 배. Park, Yongbae. Automated Testing of Reactive Software with Non-deterministic Events: A Case Study on LG Electric Oven. 비결정적 이벤트를 처리하는 반응형 소프트웨어를 위한 자동화 테스팅 기법: LG 전기 오븐 사례 연구. Department of Computer Science . 2015. 34p. Advisor Prof. Kim, Moonzoo. Text in English.

ABSTRACT

In our daily lives, we utilize numerous devices controlled by software including home appliance products such as electric ovens and refrigerators. Such devices has *reactive software* 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 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 dissertation, we report our industrial experience to solve the aforementioned problem by developing a *systematic event generation framework* based on concolic testing technique. The framework systematically generates both event ordering and event timing to test reactive software by transforming source code of the target software. The source code transformation adds a symbolic variable that decides whether or not an event is raised at a specific location in the target program. Using the transformed code, a concolic testing technique systematically generated various values on the symbolic variables to test various event ordering and timing.

We have applied the framework to a LG electric oven and detected several critical bugs in both unit testing and integration testing. In unit testing, we applied the framework to three modules where 2 concurrency bugs are found. In integration testing, we found a bug that causes the oven not response to any user inputs due to the illegal state transition. To evaluate effectiveness and efficiency of the framework, we also compared our framework with noise injection random testing technique. The comparison results show that our framework finds a corner-case bug faster than the random testing technique.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

As embedded software is prevalent in the ubiquitous computing society, ensuring correctness of embedded software becomes important. Most embedded software is reactive software as embedded software is part of a large system and interacts with other sub-systems of the large system through events. Reactive software continuously interacts with users and environment by receiving events and generating responses. Reactive software continuously processes the following tasks:

- Receiving events from users/environments through an 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).
- Updating internal state according to a given event and generating response for a given event through the main loop. For example, 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.

An event handler can receive an event any time. If an event is given while the main loop is computing its internal state, the main loop is suspended and the event handler is executed. After the event handler completes its task, the execution of the main loop resumes. The reason for executing an event handler quickly after the event arrival is to prevent from losing the event.

Due to this event-driven feature, the behaviors of reactive software depends on not only input values, but also event ordering and timing. Both the event handlers and the main loop of reactive software read and write the same memory area called shared memory. So, event ordering affects the behaviors of reactive software because the behavior of an event handler depends on contents of the shared memory which the previous execution of the event handlers may change. Similarly, event timing changes the behavior of the main loop if an event handler updates the shared memory before the main loop access the shared memory. We use a term *an event scenario* to indicate both ordering and timing of events.

Consequently, a concurrency error may occur due to race condition among the main loop and the event handlers. The execution order and timing of the main loop and the event handlers may be *non-deterministic* because events can be non-deterministically generated by users and other external entities. Incorrect synchronization on shared memory access between the main loop and the event handlers induces erroneous behavior.

Unfortunately, developers of reactive software often do not recognize this issue seriously because they think that reactive software is simple enough to be free from complex concurrency problems. The reason that reactive software in home appliance devices such as electric ovens and refrigerators runs in a single thread makes developers overlook concurrency errors in their software. So, such reactive software often suffer corner-case bugs that can be triggered by exceptional execution scenarios only.

Furthermore, it is hard to detect concurrency errors in both manual testing and automated testing by conventional techniques. The manual testing is ineffective for finding corner-case bugs because human testers are not capable of precisely controlling. Conventional testing techniques such as concolic testing do not generate various execution order/timing of events. Model checking is not effective for checking reactive software in industry due to high cost of abstract model creation and state explosion problems.

Developers in industry cannot afford time to create abstract models for reactive software due to hard time-to-market pressure. As the number of states of given model increases exponentially with the number of events, model checking may not generate outputs within limited time and memory space.

In this dissertation, we report our industrial experience to solve the aforementioned problem by developing a *systematic event generation framework*. The framework can systematically generate various event scenarios. 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 4).

We have applied the systematic 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 6.1) and an illegal state transition bug (Section 6.2) 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:

- This dissertation 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 6).
- 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 4).
- We have demonstrated the effectiveness of the event generation framework by detecting new bugs (Section 6) in an industrial reactive system (i.e., a LG electric oven).

The rest of the dissertation is organized as follows. Chapter 2 describes background and related works. Chapter 3 explains the overview of this testing project including the description of the LG electric oven. Chapter 4 explains the systematic event generation framework based on concolic testing technique. Chapter 5 describes the testing setup of the project. Chapter 6 reports the testing results. Chapter 7 discusses lessons learned from this project. Chapter 8 summarizes the dissertation and provides future works.

Chapter 2. Background and Related Works

This chapter provides background and related works. We provides concolic testing technique that our technique uses, previous testing techniques for reactive software, and their limitations.

2.1 Concolic Testing Techniques

Concolic (CONCrete + symbOLIC) testing (dynamic symbolic execution) [24, 28] runs a given program with concrete inputs, creates a symbolic path constraint from the concrete execution, and generates a new test input from analysis of the symbolic path constraints [6]. A path constraint is a condition that should be satisfied to visit a specific path. In concolic testing, the input values of the path constraint is represented as symbolic variables instead of concrete values.

2.1.1 Concolic Testing Algorithm

The inputs of concolic testing are target program code, a test driver code, and terminating condition. The output of concolic testing is test inputs and test execution results of the generated inputs. The general algorithm of concolic testing consists of the following steps.

1. Select input variables to be handled symbolically.
2. Instrument the target program to record symbolic path constraints.
3. Choose input values for the instrumented programs and run the instrumented programs using the test driver.
4. Obtain a symbolic path constraint from the execution of instrumented program.
5. Negate a branch of the obtained symbolic path constraint to create a new symbolic path constraint.
6. Find input values that satisfy the new symbolic path constraint.
7. If the given termination condition is satisfied, stop concolic testing, Otherwise, repeat step 3 with the generated input values of step 6.

In step 1, a user (or a tester) decides which input variables should be declared to be symbolic variables. A user adds API function calls of the concolic testing tool at the beginning of the test driver execution to declare symbolic variables.

In step 2, the concolic testing tool instruments both the target program and the test driver to record symbolic path constraints. The concolic testing tool statically adds probes at branching statements to record which branches are taken in program executions.

In step 3, the concolic testing tool runs the modified test driver to test the target program. If it is the first time that step 3 is performed, input values of the test driver is the given input values. Otherwise, input values from step 6 are used.

In step 4, a symbolic path constraint is created from the execution of the inserted probes. When the inserted probe is executed in step 3, the probe collects symbolic conditions of branches through dynamic analysis of the target program and creates a symbolic path constraint.

In step 5, the concolic testing tool negates a branch of the symbolic path constraint that obtained in step 4 to create a new symbolic path constraint. If the obtained symbolic path constraint has multiple branched that can be negated, the concolic testing tool selects one of them based on search strategy. For example, depth-first search strategy selects the last branch of the symbolic path constraint first and breadth-first search strategy selects the first branch of the symbolic path constraint first.

In step 6, input values that satisfy the negated symbolic path constraint is generated using SMT solvers or SAT solvers (e.g., Z3 [25], STP [10], CVC4 [2]). If the solver cannot generate input values that satisfy the symbolic path constraint, repeat step 5 to create another symbolic path constraint (i.e, negates another branch).

In step 7, the concolic testing tool checks terminating condition. A user can decide the terminating condition using the maximum number of test inputs, the maximum testing time, branch coverage, etc.

Conventional concolic testing technique generates input values but does not generates event scenarios. To test reactive software, developers may create event scenarios but this manual approach may not affordable for industrial software projects under hard time-to-market pressure. Our framework generates both input values and event scenarios with less human effort than conventional concolic testing technique because our framework automatically generates both input values and event scenarios.

2.1.2 Concolic Testing Tools

Concolic testing technique is implemented in various tools such as DART [12], CUTE [30], jCUTE [29], CREST [3], EXE [5], and KLEE [4].

DART (Directed Automated Random Testing) is a concolic testing tool for C programs. DART executes a program with random concrete inputs at the beginning to collect symbolic path constraints. After that, DART applies concolic testing technique to generate input values. A limitation of DART is that DART cannot symbolically handle dereferencing a pointer whose address aims the input.

CUTE (Concolic Unit Testing Engine) extends DART to handle data structures with pointers. CUTE uses a logical input map that represents all inputs, including memory graphs to represent the input memory graph symbolically at the beginning of an execution.

jCUTE (CUTE for Java) combines concolic testing and *race-detection and flipping algorithm* to test multi-threaded programs with data inputs. The algorithm finds two statements that are executed in different threads and access same memory location. If the execution order of the two statements can be changed, jCUTE changes the execution order to create a new thread schedule.

CREST is an open-source concolic testing tool that developed by Jacob Burnim in 2008. CREST is an extensible platform for building search strategies. CREST supports various search strategies such as DFS, random, and heuristics based on control flow graphs to explore large search space efficiently.

EXE is applied to complex system programs such as DHCP server daemons, regular expression libraries, and file system libraries that handles untyped data structure (i.e., `void*` in C language) and byte array manipulations. To perform concolic execution on the complex system code, EXE builds symbolic path constraints for all C expressions with a single bit level accuracy so that EXE handles pointers, unions, bit-fields, and bit-operators.

KLEE extends EXE to optimize performance of symbolic path constraints to handle larger programs. KLEE also handles interactions with external environments (i.e., system API calls). For example, KLEE

provides symbolic file system whose files and their contents are symbolically generated to test programs with file I/O.

2.2 Reactive Software Testing Techniques

2.2.1 Testing using Hardware Simulators

A couple of works generated event scenarios using hardware simulators to test reactive software. Regehr et al. [27] presents a technique that fires hardware events called interrupts at random times. The technique modified a hardware simulator Avrora to fire interrupts and monitor malicious behaviors. Similar to noise injection random testing, the technique may not generate buggy event scenarios with low probability.

Higashi et al. [14] extends Regehr et al.'s approach. Instead of firing interrupts at random times, the Higashi et al.'s technique fires an interrupt after a memory-access instruction is executed in a CPU emulator to find race conditions. The technique changes input values of interrupt handlers to generate buggy execution scenarios. However, the input values should be provided by users for the technique.

A limitation of testing using hardware simulators is that it is hard to apply the techniques when hardware simulators are not available. In our case study, generating event scenarios using simulators were not applicable because LGE used cheap microcontrollers whose hardware simulators do not exist.

2.2.2 Model Checking

Researchers have applied model checking techniques such as Verisoft [11] and SPIN [15] that have been applied to multi-threaded programs to find bugs in reactive programs.

Fidge et al. [9] applies model checking to an interrupt-driven avionic reactive software by manually creating state-transition model. Fidge et al. translates each basic block of target code into a transition to reduce possible states and transitions of the target software. The state of the model consists of variables for registers, program counter, current time, and interrupt time. Registers are memory locations for data and program counter represents the next basic block to be executed. The current time and the interrupt time decide when an interrupt will occur. The current time increases by 1 when a transition of a basic block is performed. The interrupt time is chosen by model checker non-deterministically. If the current time is larger than or equal to the interrupt time, the transition corresponding to the interrupt handler is performed. Otherwise, a transition of a basic block corresponding to program counter is performed. Fidge et al. used a bounded model checker SAL [26] to verify the model and found a counter example that the target software prints wrong output values.

Holzmann et al. [16] introduces a technique that mechanically extracts a state machine from annotated code of state-oriented reactive software. To recognize possible states of the target software, a user should mark conditional statements where check the current states and the occurrence of the event in the target software. From the annotated conditional statements, the technique finds possible states and actions (i.e., event handler code) that each state will perform when an event is generated. To automatically translate the code or actions into the model, the technique uses a table maps a C statement into abstracted representation of the C statement.

Chandra et al. [7] applies Verisoft to a CDMA library of Lucent Technology. Chandra et al. created a test driver to simulate the environment for the target software. The CDMA library runs on the base station of CDMA network and the CDMA library manages connections between the base station and

mobile phones. So, the CDMA library should be tested under the various behaviors of mobile phones. Chandra et al. uses non-deterministic operations (`VS_toss`) of VeriSoft to select actions among available actions in test drivers.

The limitations of these techniques for industrial application are high cost of model creation and state explosion problem. 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. Also, the number of states increases exponentially as the number of event type increases.

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. Also our approach avoids state explosion problem because our technique focuses on execution paths instead of states.

2.2.3 Interrupt Timing Analysis

Kotker et al. [21] presents a testing technique that utilizes sequential versions of interrupt-driven programs for timing analyses. Given inter-arrival time between interrupts and priority of interrupts, the technique predicts worst-case execution time of the interrupt-driven program.

Also, Yu et al. [32] introduces a framework called SimLatte that estimates worst-case interrupt latencies using a genetic algorithm. The genetic algorithm of SimLatte takes the target program and initial test cases to generate new test cases that cause significant interrupt latencies to find worst-case interrupt latencies. The initial test cases are generated by random test case generation technique. The genetic algorithm creates new test cases by crossover which switches part of input values or event scenarios of two test cases. Yu et al. applies both SimLatte and random testing technique to three embedded system applications and the results shows that SimLatte is more effective than random testing technique.

Unlike these works, 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.

2.2.4 Noise Injection Random Testing

Lei et al. [23] presents a framework for testing message-passing programs. A message-passing program consists of a collection of processes which interact each other by sending and receiving messages. So, a message-passing program is reactive software that interacts with the environment via messages. Thus, non-deterministic ordering and timing events (messages) may causes concurrency errors in message-passing programs. The framework inserts random noise before statements that generate messages. Instead of inserting random noise before every message generating statements, the framework uses coverage criteria to select locations of random noise that may increase coverage.

Similar to Lei et al., for multi-threaded program testing, Stoller [31] introduces a technique called *rstest* that inserts a statement that calls a scheduling function that would cause a context-switch to the target Java source code. The scheduling function randomly makes the choice between doing nothing or calling `sleep()` or `yield()`.

While noise injection random testing automatically generates various event scenarios, random testing may not generate event scenarios whose possibility to be generated by random testing is low and thus noise injection random testing may not generate buggy event scenarios. Our case study results show that noise injection random testing is not effective to find a corner-case concurrency error that occurs when

two events are generated at specific timing (Section 6.1). Compare to noise injection random testing, our technique generates systematically generate event scenarios to explore corner-case event scenarios faster than noise injection random testing.

Also, noise injection random testing cannot guarantee reproduce of generated event scenarios. Even if same random seed is used to produce the same noise duration, the same event scenario may not be generated because the environment such as OS, and other process that running on the same machine affects execution time.

Chapter 3. Project Overview

3.1 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) [30] 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 [17–20]. 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. The electric oven has a well-defined requirement specification for its behavior (see Figure 3.3). 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 3.3).

3.2 LG Electric Oven

Figure 3.1 shows a target LG electric oven. The target electric oven is a high-end multi-function electric oven and 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 3.3).

3.3 Electric Oven Controller Software

Figure 3.2 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.

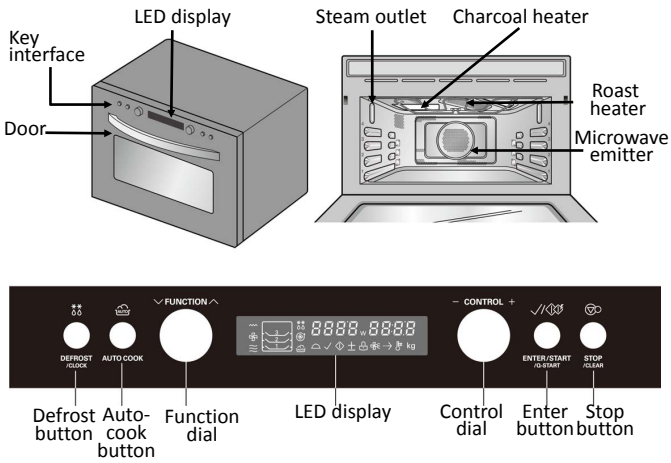


Figure 3.1: LG electric oven

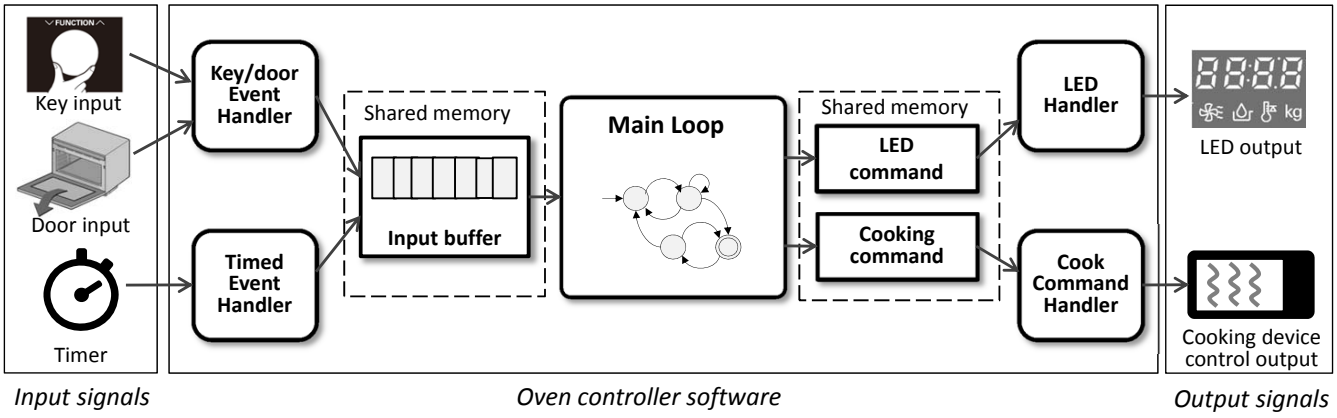


Figure 3.2: Architecture of the electric oven controller software

- *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 invoked every second [1] 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 executed every 1 milliseconds [1] and 100 microseconds [1] 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.

Figure 3.3 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

[1] To secure the intellectual property rights of LGE, the exact time intervals of the event handlers are not written in the paper.

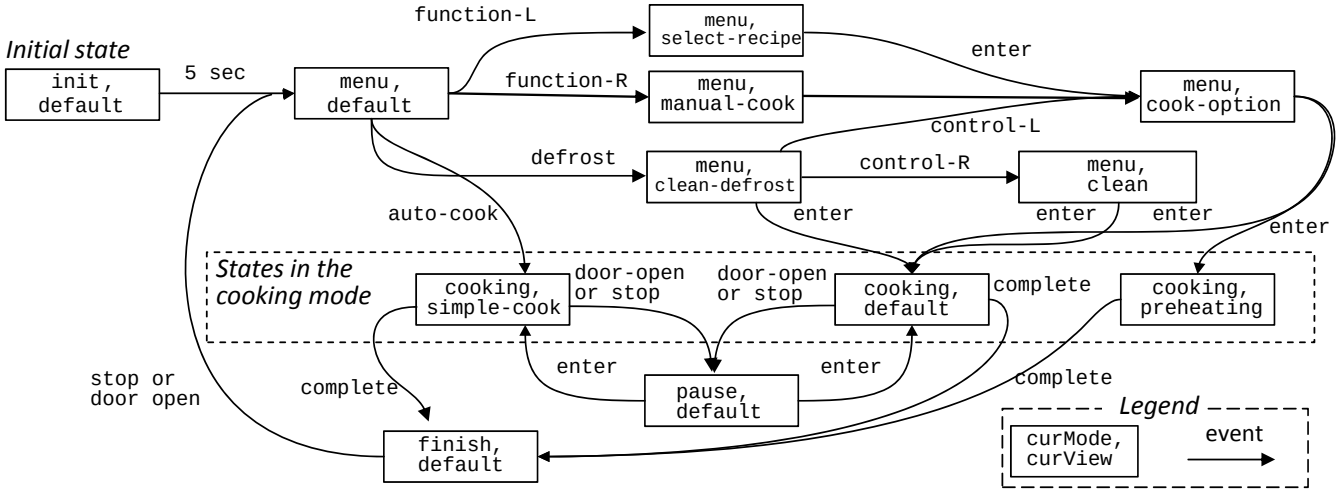


Figure 3.3: Requirement specification of the LG electric oven

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*. [2]

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

[2] The full state machine specification defines a state as a tuple of the five variables and contains 122 states and 634 transitions. Figure 3.3 is an abstract version and has non-deterministic transitions due to the abstraction.

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 runs on an 8 bit micro-controller without underlying OS. The controller software implements an event by using an interrupt signal and an event handler as a function registered for the signal.

Chapter 4. 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.

- *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.

Figure 4.1 shows the overall process of the event generation framework. The framework instruments the main loop of the target C source code by statically inserting probes at every important code location in the main loop (Section 4.1) where a probe is a function call whose parameter is a unique integer called location id representing code location of the probe. Then, the probes invoke event handlers systematically by controlling each probe through concolic testing (Section 4.2). As a result, the framework outputs both input values and event scenarios. 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 4.2).

4.1 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 4.2(b)).

We have developed the following three 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.

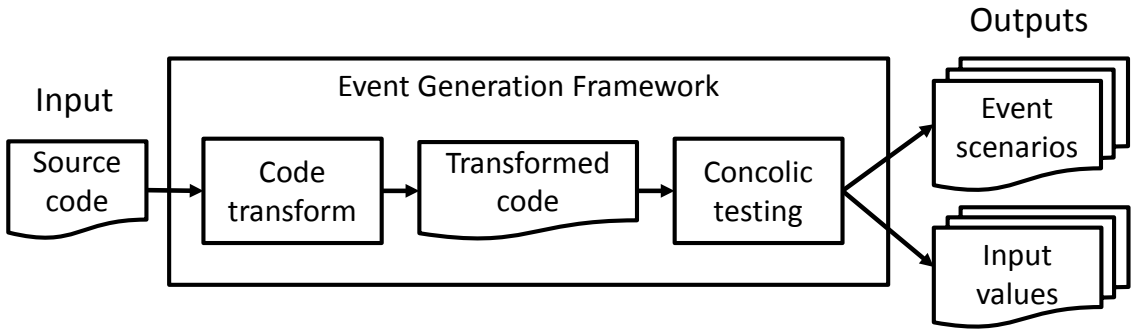


Figure 4.1: Overall process of the event generation framework

```
 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

```
 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;
11   p(1); f(y);
12   p(2); f(*ptr);}
13
14 void ev1Hdl(){
15   … ; y++; …}
```

(d) Instrumented source code by the shared variable access strategy

Figure 4.2: Example showing how the three strategies insert the probes

- *Shared variable access-based strategy* inserts the probe at every statement in the target code segment that accesses (i.e., reads or writes) a global variable shared by an event handler. This strategy conservatively assumes that a statement that uses a pointer may access a shared variable.

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.

Compared to the statement-based strategy, the shared variable access-based strategy may insert a less number of probes (because only a subset of the statements in the target segment accesses a shared variable) but achieve comparable bug detection capability. This is because the main loop and an event handler interact/interfere with each other through a shared variable and all such interferences can be enforced by the probes inserted by the shared variable access-based strategy.

Figure 4.2(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 3.3)). Figure 4.2(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 4.2(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 4.2(c) has only two basic blocks beginning at line 5 and line 10, respectively. Figure 4.2(d) shows the shared variable access based strategy. The framework inserts a probe at line 11 because `f(y)` reads a global variable `y` which is shared by an event handler `ev1Hdl()` (line 15). In addition, the framework inserts a probe at line 12 because `f(*ptr)` may dereference to a shared variable.

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 [1].

4.2 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 4.3 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 [19] to perform the dynamic symbolic execution, which is an instrumentation based dynamic symbolic execution tool for C programs (faster than KLEE [19]) (CREST-BV is the extension of CREST [3] 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 3.3). [1] If `isInHdl` is false 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

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

```
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:} } } }
```

Figure 4.3: Pseudo code of a probe for the event type `ev1`

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. For example, the LG oven has four different event types (Section 3.3)). `ev1`, `ev2`, `ev3`, and `ev4` and each probe inserted handles the above task for every event type (i.e., invoking `probe_ev1(x)`, `probe_ev2()`, `probe_ev3()`, and `probe_ev4()`). 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 6.3).

If the reactive software reads input values, additional symbolic variables should be declared to generates input values. If the reactive software reads a variable in the main loop, one dimensional array whose size is the number of maximum main loop iteration is required and each element of the array should be declared as a symbolic variable. If the reactive software reads a variable in the event handler, two dimensional array which has the same structure of `ev1Loc` is required to symbolically decide the input value for each event handler invocation.

4.3 Record and Replay of the Generated Event Scenarios

To help developers debugging the errors, the framework can record and replay the generated event scenarios. Figure 4.4 shows the pseudo code of function `init()` that record the event scenario and function `read()` that reads the recorded event scenario for replay. `init()` of Figure 4.4 contains the same code of `init()` in Figure 4.3 except `init()` of Figure 4.4 has a line that writes the contents of `ev1Loc` into a file. After that, `read()` reads the values of the file or the standard input. If the behaviors of the target program are deterministic, the behaviors of the program depends on only input values and

```
01:int ev1Loc[NUM_ITER_EV1][MAX_EV1_OCCUR];
02:
03:void init(){
04:  FIlE* outputFile = fopen(FILE_NAME, "a");
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:      fprintf(outputFile,"%d,",ev1Loc[i][j]);
09:  } }
10:  fclose(outputFile);
11:}
12
13:void read(){
14:  for(i=0;i<MAX_ITER_EV1;i++) {
15:    for(j=0;j<MAX_EV1_OCCUR;j++) {
16:      scanf("%d,",&ev1Loc[i][j]);
17:} } }
```

Figure 4.4: Pseudo code of event scenario record and replay

the event scenario. Since the framework controls both input values and event scenarios, using the same input values and event scenario (i.e., the same values of `ev1Loc`) always lead the same behavior.

Chapter 5. Testing the Oven Controller Software with the Event Generation Framework

We first applied the framework to test units of the controller (Section 5.1) and then to test the entire controller software (Section 5.2). Also, we applied noise-based random testing techniques for comparison (Section 5.3).The experiments were performed on the machine with Intel I5 4.2 GHz and 16 Gigabyte memory, which runs 64 bits Ubuntu 14.04 linux.

5.1 Unit-level Testing

We have applied the framework to the following three units:

- a circular queue (calling it CQ) to which the input event handler store input values and from which the main loop loads input values (*input buffer* in Figure 3.2).
- a load module that enforces the voltage and the current to the heaters
- a model option unit that recognizes the oven hardware and enables/disables relevant oven features.

Table 5.1 shows the size (the number of files, LOC, the number of functions) of the each module that we tested. 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. [1]

5.1.1 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 6.1) and `enqueue()` to add (store) and remove (pop up) a new value to/from the queue.

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

Table 5.1: Code metrics of units under test

Module name	# of .c files	# of .h files	LOC	# of functions
Circular queue	1	1	145	7
Load	3	3	765	33
Model option	1	1	69	6

5.1.2 Unit Testing Setup for CQ

Figure 5.1 explains how we setup the unit testing driver for CQ. `test_init()` at line 3 initializes 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 6.3). 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 5.1 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 (at line 16). 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. 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.

5.2 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 code of integration testing contains 2,807 branches.

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

```
 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:}
14:void eventHandler() {
15:  writtenData[wi++]=v;
16:  enqueue(v++);
17:}
```

Figure 5.1: Unit testing driver for CQ

Figure 3.2). We built a symbolic environment to invoke the input event handlers with various key/door events (e.g., *enter*, *stop*, *function-L*, *door-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` $\in \{2, 3, 4\}$).

We used two search strategies for concolic testing: DFS and random negation. We tested the instrumented the controller software for 1 hour per search strategy. For the random negation search strategy, we repeated the testing 30 times.

5.2.1 Test Oracles

For test oracles, the full state machine specification is utilized (see Figure 3.3 which is the abstract version of the full state transition machine). In other words, `assert()` statements are inserted at the statements that make state transitions to check if the relations of source states and the destination states are consistent with the specification. (i.e., following the state machine specification).

Figure 5.2 shows the code of the test oracles. `STATE` is data structure containing `curMode` and `curView` to represent a state. a `performTransition()` decides the next state (i.e., `performTransition()` modifies `currentState`) based on the current state (`currentState`) and the input events (`eventBuffer`). `eventBufferSize` indicates how many input events `eventBuffer` has. We inserted `checkTransition()` after the transition is performed to check whether the performed transition is correct or not. The full state machine specification is stored as a map from a source state and an event to destination states. The `getNextState()` finds the next state from the map. For example, if the source state is (*menu*, *default*) and the event is *defrost*, `getNextState()` returns (*menu*, *clean-defrost*).

```
01:STATE_MAP map;
02:STATE currentState, previousState;
02:
03:void main() {// the main loop thread
04:  initStateMap();
05:  ...
06:  while(...) {
07     ...
08:    performTransition(eventBuffer, eventBufferSize);
09:    checkTransition(eventBuffer, eventBufferSize);
10:    previousState = currentState;
11:} }
12:
12:void checkTransition(EVENT[] eventBuffer, int eventBufferSize) {
13:  STATE intermediate = previousState;
14:  for (int i = 0; i < eventBufferSize ; i++) {
15:    intermediate = getNextState(map, intermediate, eventBuffer[i]);
16:  }
17:  assert(isSameState(currentState,intermediate) == TRUE);
18:} }
```

Figure 5.2: Test oracle for the integration testing

5.3 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 [8, 22].

Figure 5.3 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. We used the Pthread signaling mechanism to mimic event handling mechanism of a reactive software (i.e., generates signals to the main thread by invoking `pthread_kill(main_loop_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 and 5, 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. [2]

[2] An event generation thread takes around 65 microseconds on average to execute `usleep()` and `pthread_kill()` (lines

```
01:int main() {// the main loop thread
02:  ...
03:  while(...) {
04:    delay(200); stmt1();
05:    delay(200); stmt2();
06:} }
07:
08:void evHnd() { ... }
10:void *evGen(...){//the event generation thread
11:  while(...) {
12:    usleep(rand() % MAX_EVENT_GEN_SLEEP);
13:    pthread_kill(main_loop_thread...);
14:} }
```

Figure 5.3: Noise injection based random testing

- *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 12 can be 500, 1000 or 1500). The average value of `rand()%MAX_EVENT_GEN_SLEEP` is `MAX_EVENT_GEN_SLEEP/2` because the distribution of `rand()` is uniform.

We selected 500 microseconds as granularity of timing delays because the average value of `rand()%500` is 250 which is similar to 200. So the timing delay of maximum 500 microseconds has a high probability of generating an event at every statement and the timing delay of maximum 1000 microseconds has a high probability of generating an event at every other statement. We used the current time-stamp as random seed.

For the unit testing of CQ, we used the similar unit testing driver in Figure 5.1 except that CQ is initialized randomly not symbolically. We tested CQ for 30 minutes and repeated the random testing 30 times.

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.

12-13 of Figure 5.3) 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.

Chapter 6. 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.

6.1 Results of the Unit Testing

We detected the following two atomicity violation bugs in CQ (Section 5.1) by using the event generation framework [1].

- an *overwriting bug* which causes the queue to overwrite the oldest value in the queue with a new value and cause `dequeue()` (see Figure 6.1) 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 6.1 describes a simplified `dequeue()` of the circular queue which has these two bugs.

Figure 6.2 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 5.1). 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 the input event handler which calls `enqueue()` is invoked between line 6 and line 8 of `dequeue()` when the queue is full.

Figure 6.3 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).

[1] The overwriting bug and the inconsistency bug are *multi-variable* atomicity violation bugs [13] 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.

```
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; }
```

Figure 6.1: Buggy `dequeue()` of the circular queue CQ

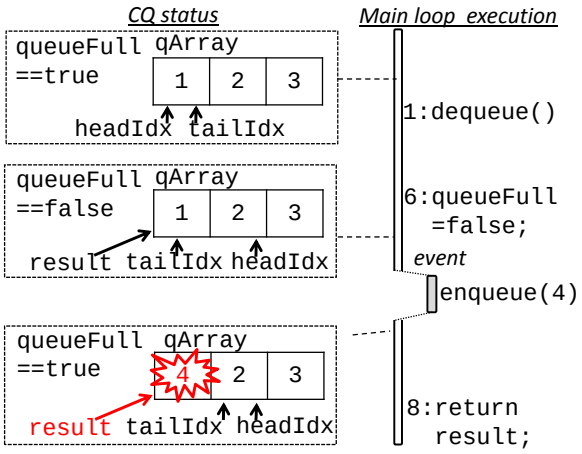


Figure 6.2: Error caused by the overwriting bug

Table 6.1 shows the testing results of the event generation framework and the random techniques. The systematic event generation framework inserted 17, 11, and 12 probes in CQ by the statement-based, basic block-based, and shared variable access-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 6.1 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 12th 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 eighth and 10th 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

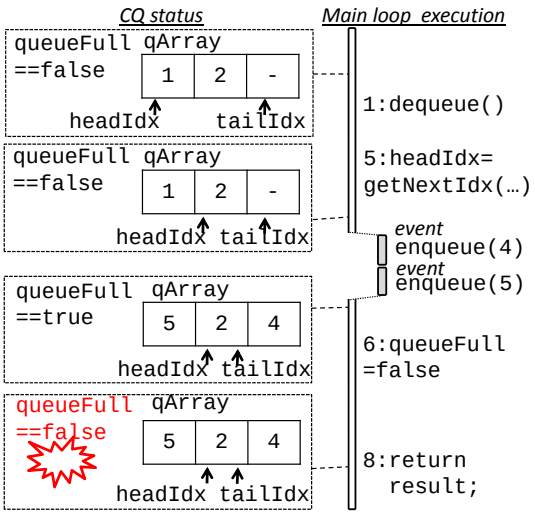


Figure 6.3: Error caused by the inconsistency bug

Table 6.1: Time to detect the bugs in Circular Queue

	Event generation framework						Random testing					
	Statement		Basic Block		Shared Var. Acc.		0–500 μsec.		0–1000μsec.		0–1500 μsec.	
	DFS	Random	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	0.07 (2)	0.33 (25.70)	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	40.37 (2674)	320.67 (19.41K)	0.20	1653.00 (0.33M)	0.23	1577.90 (0.31M)	0.00	1800.00 (0.36M)

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 random techniques generates redundant execution scenarios repeatedly which wastes testing time. We say the two execution scenarios are same if their initial CQ states and event scenarios (i.e., event timing) are same. The percentages of redundant execution scenarios in maximum 500, 1,000, and 1,500 microseconds delay are 21.6%, 26.5%, and 31.1%, respectively. 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.

6.2 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.

```
01:int curMode, curView;
02:int nextMode, nextView;
03:void main() {
04:  while (1) {
05:    curMode = nextMode;
06:    curView = nextView;
07:    int keyCnt = GetKeySize();
08:    for (int i = 0; i<keyCnt; i++)
09:      KeyHandler(GetKey());
10:} }
11:void KeyHandler(int keyId) {
12:  switch (curMode) {
13:    case MENU:
14:      switch (curView) {
15:        case DEFAULT:
16:          switch (keyId) {
17:            case DEFROST:
18:              nextView=CLEAN_DEFROST;
19:              break;
20:            case AUTO_COOK:
21:              nextMode=COOKING;
22:              nextView=SIMPLE_COOK;
23:              break;
24:    ...
25:} } }
```

Figure 6.4: Buggy `KeyHandler()` code of the oven control software

This illegal transition was made by the two consecutive events *auto-cook* and *defrost* 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 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 `KeyHandler()` which makes *multiple state transitions* in one main loop iteration. Using `KeyHandler()`, the controller can handle multiple events fast in one main loop iteration. But this makes the controller program complicated and `KeyHandler()` does not operate correctly with unexpected event sequences.

Figure 6.4 shows the simplified main loop and `KeyHandler()`. The main loop of the oven software handles multiple key inputs in a single main loop iteration to provide better responsiveness to users. `curMode` and `curView` represent the current state (Section 3.3) and `nextMode` and `nextView` represent the next state of the oven control software. `nextMode` and `nextView` are modified in `KeyHandler()` and the next state becomes the current state after line 5 and line 6 are executed. After the current state is changed, the main loop reads key data from the input event buffer and calls `KeyHandler()` for

Table 6.2: Time to detect the bug in the controller program and branch coverage

Techniques			Detect. time	# of exec.	Branch coverage
Event generation framework	$n = 2$	STMT	195.10	3991.40	50.2%
		BB	172.63	3800.33	50.0%
		SVA	113.63	2932.10	50.9%
	$n = 3$	STMT	208.60	2920.60	51.8%
		BB	249.43	3845.10	51.7%
		SVA	150.47	2858.57	52.1%
	$n = 4$	STMT	280.03	3210.60	52.6%
		BB	249.63	3307.43	52.8%
		SVA	197.93	2594.03	53.4%
Random testing		500 μsec.	22.63	1.77	55.4%
		1000 μsec.	23.63	1.97	54.9%
		1500 μsec.	24.30	1.47	54.3%

each key input event. `KeyHandler()` has nested switch statements to perform actions depending on the current state which is represented by `curMode` and `curView`, and a key input. When the current state is (*menu*,*default*) and the input event buffer has a key input *defrost*, for example, line 18 of `KeyHandler()` is executed and the next state is (*menu*,*clean-defrost*).

In the above erroneous execution, `KeyHandler()` updates the current state from (*menu*,*default*) to (*cooking*,*simple-cook*) with *auto-cook* event first (see Figure 3.3) by executing line 20-23 of 6.4. Then, with *defrost* event, `KeyHandler()` incorrectly updates the current state based on the previous state (i.e., (*menu*,*default*)), not the recently updated state (i.e., (*cooking*,*simple-cook*) at line 18 of Figure 6.4. Since some event such as *defrost* may update the current state partially (i.e., updating only `curView`, not `curMode`), `KeyHandler()` updates `curView` to *clean-defrost* with *defrost* (as shown at the top of Figure 3.3) because `KeyHandler()` think that the current state is still (*menu*,*default*). As a result, `KeyHandler()` updates the current state as (*cooking*,*clean-defrost*) 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 `KeyHandler()`, all assert violations were removed.

Table 6.2 shows the testing results of the event generation framework and the random techniques on the controller software. The systematic event generation framework inserted 5,009, 2,339, and 1,294 probes by the statement-based, basic block-based, and shared variable access-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 shared variable access-based strategy (SVA) 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 113.63 seconds after executing the controller software 2932.10 times on average (see the second row of the table). Also, we can observe that SVA is faster than STMT and BB to detect the bug for all n values. In addition, the bug detection time increases as `MAX_EV_OCCUR` increases from 2 to 4 (i.e., from 113.63 seconds to 197.93 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 covered 50-55% of the branches of the target

Table 6.3: The number of executed probes and length of path constraints in concolic testing

Techniques		# of executed probes	Path constraint length
$n = 2$	STMT	19419.04	940.33
	BB	16129.21	1072.30
	SVA	12786.93	1522.93
$n = 3$	STMT	24875.50	1600.86
	BB	20937.96	1879.20
	SVA	17326.37	2969.26
$n = 4$	STMT	28211.48	2452.70
	BB	23919.24	2953.03
	SVA	20127.40	4760.46

controller software. The random testing achieves higher branch coverage than event generation framework because random testing calls event handler more frequently than the event generation framework. The number of each event handler invocations is 943.8 per main loop iteration in average.

The random testing techniques detected the bug five times faster than the event generation framework with the shared variable access-based strategy (SVA) 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 fifth 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.

Table 6.3 shows the number of executed probes per execution and length of symbolic path constraints in average. The number of executed probes with SVA is lower than the number of executed probes with the other strategies because SVA inserts fewer probes than the other strategies. We observed that executed paths of SVA is longer than the other strategies because concolic testing with SVA negates more input-related branches than the other strategies. Concolic testing with SVA have more probability to negate input-related branches than the other strategies because the number of executed probes with SVA is low.

Figure 6.5 shows branch coverage of STMT per $n \in \{2, 3, 4\}$ where n is the maximum number of events per the main loop iteration. The horizontal axis represents the execution time, and the vertical axis represents branch coverage. At the beginning of the test, the branch coverage of small n is higher than that of large n because the testing with small n quickly covers branches due to shorter execution time. After certain point, however, the testing with large n achieves higher coverage than that of small n because the testing with large n generates event scenarios that the testing with small n. This phenomenon also appears in BB and SVA.

Figure 6.6 shows branch coverage per probe insertion strategy (STMT, BB, SVA) where n (the maximum number of events per the main loop iteration) is 4. The graph shows that SVA is more efficient than BB and STMT because the branch coverage of SVA is always higher than that of BB and STMT. SVA also achieves higher branch coverage than BB and STMT when $n = 2$ and $n = 3$.

Figure 6.7 represents 2 venn diagrams to show the number of covered branch per each strategy

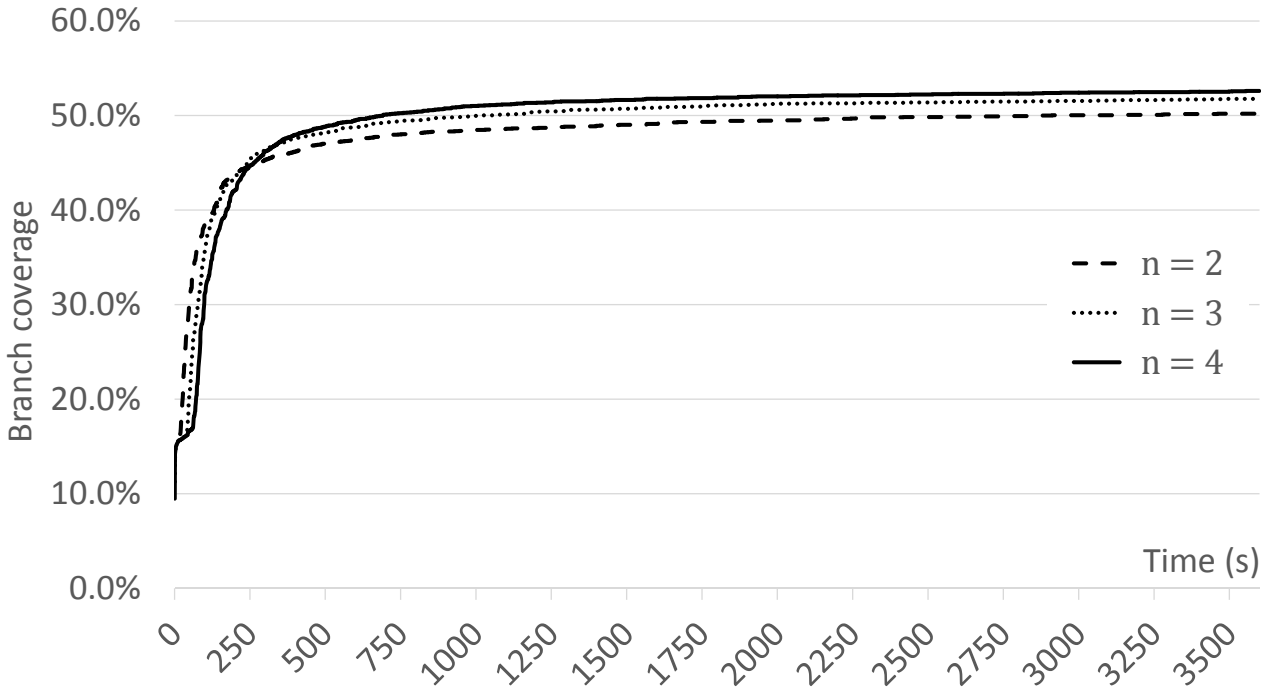


Figure 6.5: Branch coverage of STMT for each maximum number of events per the main loop iteration

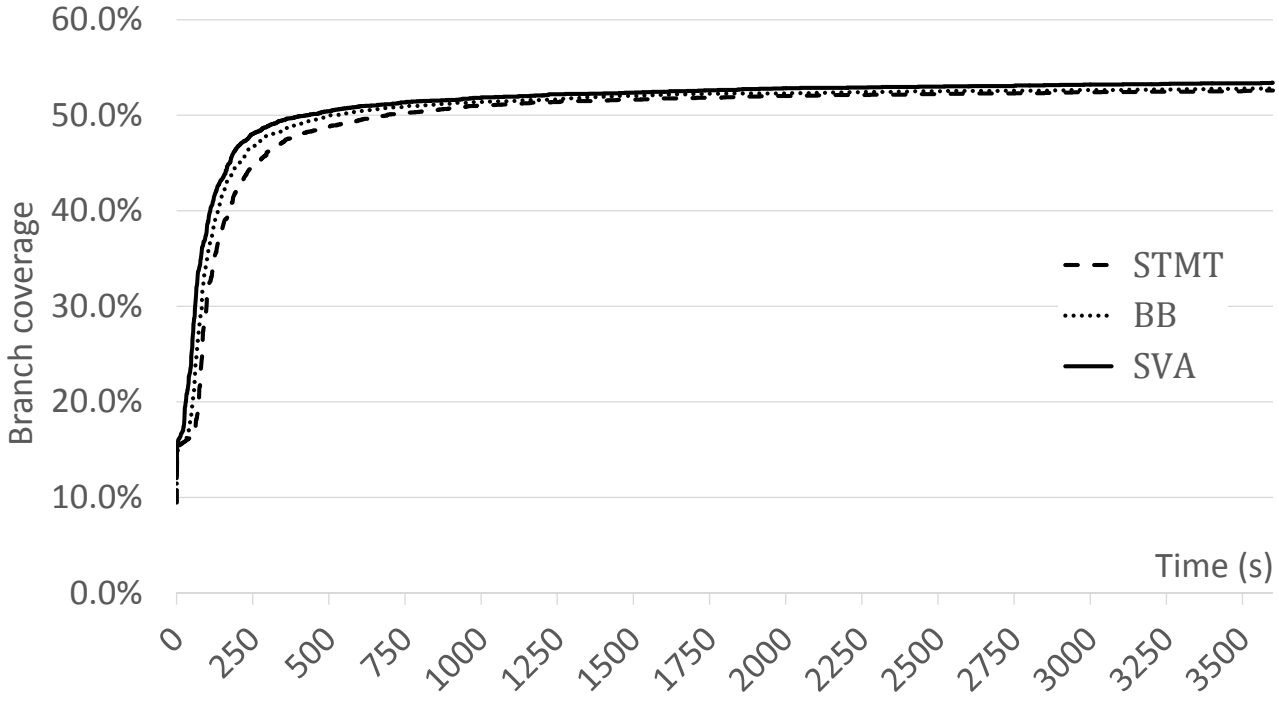
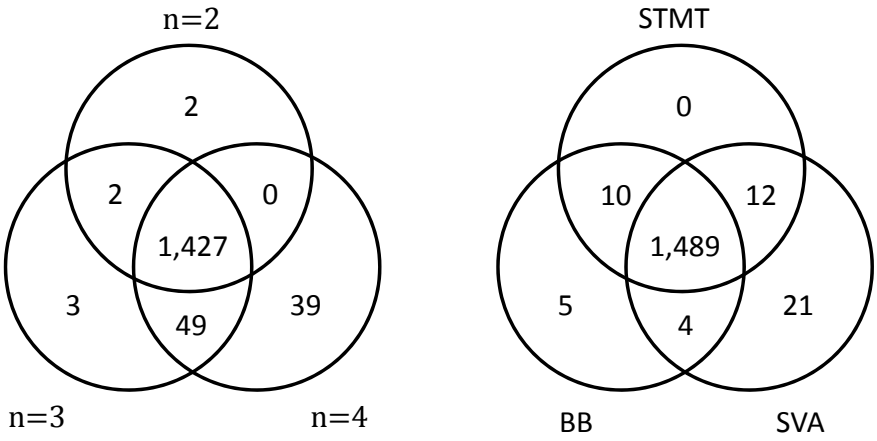


Figure 6.6: Branch coverage per probe insertion strategy when $n = 4$

and each $n \in \{2, 3, 4\}$. Figure 6.7(a) shows the number of covered branches per maximum number of events per the main loop iteration. 88 branches are not covered when $n = 2$ but they are covered when $n = 4$. Figure 6.7(b) shows the number of covered branches per probe insertion strategy. SVA covers 21 branches that STMT or BB cannot cover and missed 15 branches that STMT or BB covered. There is



(a) Covered branches of STMT for each $n \in \{2,3,4\}$

(b) Covered branches for each probe insertion strategy when $n = 4$

Figure 6.7: Venn diagram of the number of covered branches in integration testing

no branch that only STMT can cover.

Chapter 7. Lessons Learned

7.1 Effectiveness of the Event Generation Framework

Through the project, we confirmed that the event generation framework can detect critical corner-case bugs effectively (Section 6). 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 4). Thus, by applying the framework, developers can effectively improve the quality of industrial reactive software.

7.2 Systematic Testing vs. Random Testing

We have compared the event generation framework with the carefully designed random testing techniques (Section 5.3). 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 6.2).

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.

7.3 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.

7.3.1 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.

7.3.2 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.

7.4 Technical Challenges

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

7.4.1 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 6.2). 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.

7.4.2 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.

Chapter 8. Conclusion and Future Work

8.1 Summary

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. To enable systematic testing of various event handler execution scenarios, we have developed an automated event generation framework that inserts probes in the main loop of the target reactive program and selects probes that calls the event handlers using symbolic variables of concolic testing. We applied the framework to a LG electric oven controller software and detected several critical errors in the controller software, which had not been detected by the field engineers before. In addition, the comparison with random testing technique shows that our framework is more effective to find corner-case bugs. 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.

8.2 Future Works

8.2.1 Improving Efficiency

We will apply static def-use analyses to reduce the number of inserted probes to improve efficiency of the framework. We found that the number of inserted probes are important factor of efficiency (Section 6.2. In our approach, shared variable access-based strategy inserts probes at every statement that dereferences pointer because we assume that every pointer dereference accesses a shared variable. Def-use analyses may reduce the number of probes if we do not insert probes at a statement that dereferences a pointer of non-shared variables.

8.2.2 Improving Effectiveness

The implementation of the probe of our approach cannot distinguish multiple invocations of the probe in the same main loop iteration and declaring additional symbolic variables to distinguish multiple probe invocations generate more fine-grained event scenarios than the current approach. The first invocation of the probe can call the event handlers but the next invocations of the same probe cannot call the event handlers because the probe removes all elements with ID of the invoked probe from `ev1Loc`. We will change the condition of calling the event handler so that the condition become true if the number of previous invocations of the probe and the values of symbolic variables are same to distinguish multiple invocations of the same probe.

References

[1] Clang: a C language family frontend for LLVM. `http://clang.llvm.org`.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proc. of Int. Conf. on Computer Aided Verification (CAV)*, 2011.

[3] J. Burnim. CREST - automatic test generation tool for C. `http://code.google.com/p/crest/`.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of USENIX Conf. on Operating System Design and Implementation (OSDI)*, 2008.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[6] C. Cader, P. Godefroid, S. Khurshid, C. S. Pâsâreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice - preliminary assessment. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, 2011.

[7] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: An industrial case study. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, 2002.

[8] 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.

[9] C. Fidge and P. Cook. Model checking interrupt-dependent software. In *Proc. of Asia-Pacific Software Engineering Conference (APSEC)*, 2005.

[10] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. of Int. Conf. on Computer Aided Verification (CAV)*, 2007.

[11] P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *Proc. of Int. Conf. on Computer Aided Verification (CAV)*, 1997.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of ACM SIGPLAN on Programming Language Design and Implementation (PLDI)*, 2005.

[13] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set serializability violations. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, 2008.

[14] M. Higashi, T. Yamamoto, and Y. Hayase. An effective method to control interrupt handler for data race detection. In *Proc. of Workshop on Automation of Software Test (AST)*, 2010.

[15] G. Holzmann. *The Spin Model Checker*. Wiley, New York, 2003.

[16] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, 1999.

[17] 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.

[18] M. Kim, Y. Kim, and Y. Jang. Industrial application of concolic testing on embedded software: Case studies. In *Proc. of Int. Conf. on Software Testing, Verification and Validation (ICST)*, 2012.

[19] 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 *Proc. of Int. Conf. on International Conference on Software Engineering (ICSE) SEIP track*, 2012.

[20] 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 *Proc. of Int. Conf. on Automated Software Engineering (ASE) experience track*, 2013.

[21] J. Kotker, D. Sadigh, and S. A. Seshia. Timing analysis of interrupt-driven programs under context-bounds. In *Proc. of Int. Conf. on Formal Methods in Computer Aided Design (FMCAD)*, 2011.

[22] B. Křena, Z. Letko, T. Vojnar, and S. Ur. A platform for search-based testing of concurrent software. In *Proc. of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2010.

[23] Y. Lei and E. Wong. A novel framework for non-deterministic testing of message-passing programs. In *Proc. of Int. Symposium on High-Assurance Systems Engineering (HASE)*, 2005.

[24] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, 2007.

[25] L. Moura and N. Bjorner. Z3: An efficient SMT solver. In *Proc. of Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[26] L. Pike. Model checking for the practical verificationist: a user's perspective on SAL. In *Proceedings of the Automated Formal Methods Workshop (AFM)*, 2007.

[27] J. Regehr. Random testing of interrupt-driven software. In *Proc. of ACM Int. Conf. on Embedded Software (EMSOFT)*, 2005.

[28] K. Sen. Concolic testing. In *Proc. of Int. Conf. on Automated Software Engineering (ASE)*, 2007.

[29] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *Proc. of Int. Conf. on Computer Aided Verification (CAV)*, 2006.

[30] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2005.

[31] S. D. Stoller. Testing concurrent java programs using randomized scheduling. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 70(4):142–157, 2002.

[32] T. Yu, W. Srisa-an, M. B. Cohen, and G. Rothermel. Simlatte: A framework to support testing for worst-case interrupt latencies in embedded software. In *Proc. of Int. Conf. on Software Testing, Verification and Validation (ICST)*, 2014.

Summary

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

일상 생활 속에서 사용하는 자주 사용하는 전자레인지, 냉장고와 같은 가전제품에는 하드웨어를 제어하는 반응형 소프트웨어가 포함되어 있다. 반응형 소프트웨어는 이벤트 핸들러를 실행하여 사용자 입력과 이벤트를 받고, 입력 받는 이벤트에 따라서 내부 상태를 변경한 후에, 결과를 출력하는 것을 반복한다. 이러한 반응형 소프트웨어에서는 이벤트의 발생이 *비결정적(non-deterministic)*이며, 이벤트 핸들러와 메인 루프의 실행 순서가 경쟁 상태이기 때문에 개발자가 의도하지 않은 이벤트 발생 순서에 의해서 오류가 발생할 수 있다. 따라서 반응형 소프트웨어를 효과적으로 테스팅하기 위해서는 사용자 입력 값에 따라서 소프트웨어가 올바르게 동작하는 가를 확인하는 것뿐만이 아니라, 다양한 이벤트 발생 순서와 발생 시점에 따라서 소프트웨어가 올바르게 동작하는 가를 확인해야 한다. 즉, 반응형 소프트웨어를 테스트할 때, 메인 루프의 실행을 기준으로 한 상대적인 이벤트 발생 시점과 이벤트의 순서를 제어하여 소프트웨어의 다양한 행동을 관찰하는 것이 중요하다.

본 논문은 산업체에서 개발하는 반응형 소프트웨어를 테스트하기 위해서 concolic 테스팅 기법을 기반으로한 *이벤트 생성 프레임워크*를 소개하고, 실제 반응형 소프트웨어에 적용한 사례연구를 설명한다. 이 프레임워크는 실행 이벤트의 발생 순서와 시간을 체계적으로 생성하기 위해서 반응형 소프트웨어의 소스 코드를 수정하고, 수정된 코드에 대해서 concolic 테스팅을 수행한다. 코드 수정 단계에서는 심볼릭 변수를 추가하고, 대상 프로그램의 특정 실행 시점에 이벤트를 발생시킬지를 심볼릭 변수로 결정한다. 이후 수정된 코드로 concolic 테스팅을 수행하면, concolic 테스팅은 심볼릭 변수를 체계적으로 변경하면서 다양한 이벤트 발생 시점과 순서를 생성한다.

이벤트 생성 프레임워크를 LG전자의 전기 오븐에 적용하여 단위 테스트와 통합 테스트를 수행한 결과, 오븐 제어 소프트웨어 내의 버그를 찾아내었다. 단위 테스트에서는 오븐 제어 소프트웨어의 3개의 모듈에 프레임워크를 적용하였고, 2개의 동시성 버그를 찾아내었다. 통합 테스트에서는 오븐이 비정상적인 상태로 전환되어 사용자의 입력이 무시되는 버그를 찾아내었다. 그리고 프레임워크가 얼마나 효과적이고 효율적인가를 실험적으로 보이기 위해서 랜덤 테스팅 기법과 비교하였으며, 프레임워크가 렌덤 테스팅보다 발견하기 어려운 버그를 더 효과적이고 효율적으로 찾는다는 것을 실험적으로 보였다.

감 사 의 글

2년간의 석사 과정 동안, 제가 이 논문을 완성하는데 도움을 주신 모든 분들께 이 지면을 빌어 감사의 말씀을 전하고자 합니다. 먼저 어려운 때가 있을 때마다 격려 해준 아버지, 어미니, 동생에게 감사 드립니다. 바쁘다는 핑계로 제가 자주 전화 드리지 못하였지만, 부모님께서는 저에게 자주 전화를 해주셨고, 제가 어려울 때 격려 해주신 것이 큰 힘이 되었습니다.

두 번째로 저에게 가르침을 주신 김문주 교수님께 진심으로 감사 드립니다. 2년간의 석사과정 기간 동안 흔들리는 저를 바로잡아 주시고, 올바른 길로 인도해 주신 덕분에 연구와 소프트웨어 자동화 테스팅에 대해서 잘 배울 수 있었습니다. 그리고 제가 연구하는 동안 작은 결점 하나 놓치지 않도록 도와주시고, 저의 부족한 부분을 채워 주신 덕분에 ICSE에 제출할 수준의 논문을 쓸 수 있었습니다. 또한 동시성이라는 어려운 분야를 이해하는데 도움을 주시고, 논문을 쓰는 과정에서 함께 고민해주신 홍신 박사과정과 concolic 테스팅에 대해서 저에게 아낌없는 조언을 해주신 김윤호 박사과정에게 감사 드립니다. 그리고 사례 연구를 진행하는 동안 많은 도움을 주신 LG전자의 조준희 주임님, 이동주 주임님, 장훈 주임님께 감사 드립니다.

세 번째로 석사 과정을 먼저 졸업하시면서 저에게 다양한 길을 알려주신 김영주, 안재민, 문석현 졸업생께 감사 드립니다. 저의 석사 과정 동안에 조언해 주신 것과 더불어서, 함께 있으면서 연구를 어떻게 하는가에 대해서 배울 수 있었습니다. 그리고 글을 짜임세 있게 쓰는데 도움을 주셨던 문영주 박사님께도 감사 드립니다. 또한 연구실에서 함께 생활한 연광흠 석사과정과 연구실 후배인 곽태훈, 김태진 학생에게 감사 합니다.

마지막으로 2년 동안 함께 있어준 친구, 동아리 선후배, 경기북고등학교 동창에게 감사 드립니다. KAIST에서 생활하는 동안에 다양한 분야에 대해서 보고 들으면서 시야를 넓히는데 도움이 되었습니다. 그리고 공부와 연구를 위한 훌륭한 환경을 제공해 주신 KAIST 전산학과의 여러분께 감사 드립니다.

여러분께서 도와주신 만큼, 이 논문이 다른 사람에게 큰 도움이 되었으면 합니다.

이 력 서

이 름 : 박 용 배

생 년 월 일 : 1989년 4월 3일

출 생 지 : 경기도 부천시

본 적 지 : 경기도 수원시 장안구 조원동 한일타운 105동 406호

주 소 : 경기도 수원시 장안구 조원동 한일타운 105동 406호

E-mail 주 소 : yongbae2@gmail.com

학 력

2005. 3. – 2007. 2. 경기북과학고등학교 (2년 수료)

2007. 3. – 2013. 2. 한국과학기술원 전산학과 (B.S.)

2013. 3. – 2015. 2. 한국과학기술원 전산학과 (M.S.)

경 력

2009. 6. – 2009. 8. LG전자 서초CTO 인턴십 (DTV 원격 제어 시스템)

2013. 3. – 2013. 6. CS204 Discrete Math 조교

2013. 9. – 2013. 12. CS453 Automated Software Testing 조교

학 회 활 동

1. **Y. Park**, Y. Kim and M. Kim, *A Comparative Study of Static Analysis Tools:A Case Study on libexif by Using Coverity and Sparrow*, Proceedings of the 39th KIISE Fall Conference, vol. 39, no. 2, pp. 55-57, 2012 (Best paper presentation award).
2. **Y. Park**, Y. kim, J. Cho and M. Kim, *심볼릭 라이브러리를 이용한 효과적인 Concolic 테스팅*, Korea Conference on Software Engineering (KCSE), 2014, (Best paper award).
3. S. Hong, **Y. Park**, and M. Kim, *Detecting Concurrency Errors in Client-side JavaScript Web Applications*, IEEE International Conference on Software Testing, Verification and Validation (ICST), 2014 (Acceptance rate: 28%).

연 구 업 적

1. Y. Kim, **Y. Park**, and M. Kim, *A Comparative Study of Static Analysis Tools through a Case Study*, Journal of KIISE: Computing Practices and Letters, Vol 19, Num 8, Aug 2013.
2. **Y. Park**, S. Hong and M. Kim, *Performance Bug Detection in Web Applications through Cross-browser Profiling*, Journal of KIISE: Computing Practices and Letters, Vol. 19, Num. 11, Nov. 2013 (Undergraduate award).

수 상 실 적

1. WAVE: Testing Framework to Detect Concurrency Bugss in Dynamic Web Application, Qualcomm Innovation Award 2013, 2013
2. 조합적 동시성 커버리지를 이용한 효과적인 동시성 프로그램 테스트 생성, Best paper award at KIISE's 33rd student research paper competition (graduate student track), 2014