Automated Unit Testing of Large Industrial Embedded Software using Concolic Testing

Yunho Kim
CS Dept. KAIST
South Korea
Email: kimyunho@kaist.ac.kr

Youil Kim, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang
Samsung Electronics
South Korea
Email:{youil.kim,taeksu.kim,gunman.lee,yoonkyu.jang}@samsung.com

Moonzoo Kim
CS Dept. KAIST
South Korea
Email: moonzoo@cs.kaist.ac.kr

***Abstract*—Current testing practice in industry is often ineffective and slow to detect bugs, since most projects utilize manually generated test cases. Concolic testing alleviates this problem by automatically generating test cases that achieve high coverage. However, specialized execution platforms and resource constraints of embedded software hinder application of concolic testing to embedded software. To overcome these limitations, we have developed CONcrete and symBOLic (CONBOL) testing framework to unit test large size industrial embedded software automatically. To address the aforementioned limitations, CONBOL tests target units on a host PC platform by generating symbolic unit testing drivers/stubs automatically and applying heuristics to reduce false alarms caused by the imprecise drivers/stubs. We have applied CONBOL to four million lines long industrial embedded software and detected 24 new crash bugs. Furthermore, the development team of the target software adopted CONBOL to their development process to apply CONBOL to the revised target software regularly.**

I. Introduction

Software testing is a de facto standard method to assess the quality of software. However, current testing practice in industry often fails to detect bugs in target programs, because it is difficult for human engineers to manually write effective test cases to explore specific execution paths that trigger hidden bugs. Also, manually writing effective test cases is a time-consuming task and it is difficult to write sufficient number of test cases in a limited project time.

To solve these problems, we have applied concolic (CONCrete + symbOLIC) testing techniques to industrial software projects. Concolic testing [32] (also known as dynamic symbolic execution [34] and white-box fuzzing [12]) combines concrete dynamic analysis and static symbolic analysis to automatically generate test cases to explore various execution paths of a target program. We have applied concolic testing to several industrial projects (e.g., flash memory device driver [14], mobile phone software [15], and `libexif` library which manipulates image meta data [18]) and confirmed that concolic testing is effective for detecting bugs in industrial software with modest effort.

However, since concolic testing depends on a target execution environment, specialized execution platforms and resource constraints of embedded software hinder application of concolic testing to embedded software due to the following challenges:

- A concolic testing tool should be ported to a target embedded OS (VxWorks, QNX, etc), since most concolic testing tools are developed to run on Linux or Windows. In addition, libraries that concolic testing tools utilize (e.g., parsers and SMT solvers) should be ported, too. Furthermore, source code of concolic testing tools and relevant libraries may not be available, which even disables porting.
- Concolic testing can generate millions of test cases to explore many possible execution paths of target embedded software. Running a large number of test cases and solving corresponding symbolic path formulas on resource-constrained embedded hardware may take a large amount of time which is unacceptable in industrial projects.
- For concolic testing tools targeting source-code such as CREST-BV [18], concolic testing depends on a target code build environment. Embedded programs often contain non-standard programming language extensions and require a specific compiler/linker for a target embedded platform, which may not be supported by concolic testing tools.

To address the aforementioned challenges, we have developed CONcrete and symBOLic (CONBOL) testing framework that generates symbolic unit testing drivers/stubs and performs concolic testing on a host PC automatically. Conventional unit testing assumes that the developers who have built target units perform unit testing and they can make unit testing drivers/stubs correctly and specify requirement properties based on their knowledge of the target units [27]. However, this assumption is often *not* true in industrial projects where developers have difficulty to perform unit test due to tight project schedule (for example, smartphone products are often developed in less than six months due to heavy market competition). CONBOL handles the aforementioned tasks (i.e., to make unit testing driver/stub code and specify assertions) *automatically*. CONBOL has additional benefits for testing embedded software by applying concolic testing at unit level since unit testing has less dependency on the target hardware platform by building/utilizing testing driver/stubs [20].

CONBOL targets *crash bugs* (i.e., null pointer dereferences, illegal memory accesses, and divide-by-zero bugs), because these crash bugs can cause entire software to fail and, thus, can be of important concern to developers. In addition, oracles/assertions for these bugs are readily available without user-specified assertions and can be checked automatically. In addition, CONBOL generates *symbolic unit test drivers and stubs* automatically, which enable automated unit testing of embedded software and save a large amount of human effort. Furthermore, CONBOL applies several heuristics to reduce false alarms caused by the imprecise test driver/stubs (see Sections III-B– III-D). [1]

We have applied CONBOL to four million lines long industrial embedded software for smartphones (calling it 'project S' in this paper), which contains around 49,000 functions. After filtering out false alarms, CONBOL detected 24 new crash bugs which had not been detected by manual testing nor static analysis tools such as Coverity Prevent. Furthermore, the development team of the project S recognized the practical usefulness of CONBOL and decided to adopt CONBOL in their development process to apply CONBOL to the revised code regularly. To our knowledge, this is the first case study that demonstrates the practicality of concolic testing as an automated crash bug detection technique for multi-million lines long industrial embedded software.

The remainder of this paper is organized as follows. Section II explains the CONBOL framework. Section III explains the heuristics used to improve the effectiveness and precision of bug detection. Section IV describes the case study of applying CONBOL to the large industrial embedded software project of Samsung Electronics. Section V reports the testing results and Section VI discusses lessons learned from this study. Section VII explains related work and, finally, Section VIII summarizes the paper and discusses future work.

II. CONBOL Framework

A. CONBOL Overview

Figure 1 shows the overall structure of the CONBOL framework. CONBOL is developed based on a concolic testing tool CREST-BV [18], and consists of the following six components - *CONBOL Trim*, *CONBOL Gen*, *CONBOL Pre-processor*, *CONBOL Instrumentor*, *CONBOL Library*, and *CONBOL Run*. The first three components are newly developed modules from scratch and the last three components are the extension of CREST-BV. The CONBOL framework consists of 5500 LOC in the Ruby scripting language, 5600 LOC in Ocaml for additional instrumentation, and about 8500 LOC in C/C++ to implement CONBOL main engine and modify CREST-BV's symbolic execution engine to support symbolic array index dereference by using memory model [9]. CONBOL has been developed by three Samsung engineers for five months.

The work-flow of CONBOL is as follows. Given target C code written for an embedded platform is transformed to GCC compatible C code by CONBOL Trim. Then, by analyzing this GCC compatible target C code, CONBOL Gen generates unit test driver/stub code for a target unit function. At the same time, CONBOL Pre-processor inserts `assert()` to detect crash bugs more effectively and constraints to satisfy pre-conditions of a target unit to reduce false alarms. This pre-processed GCC compatible target C file is instrumented and compiled with the unit test driver/stub code and the CONBOL library by `gcc`. Finally, the generated Linux binary file is executed by CONBOL Run to explore various execution paths and report violated assertions having high scores at run time. From the run time execution information, CONBOL reports crash bug detected and branch coverage achieved so far.

[1]For example, CONBOL may report a null-pointer dereference bug for a pointer parameter `pt` in a target unit `void f(int *pt) { ... if(*pt==10) ...}`. However, if `f()` is always invoked with `pt` as a non-null value in the target program, this bug report becomes a false alarm.

B. CONBOL Trim: Automated Porting of Unit Functions Written for an Embedded Platform

CONBOL Trim removes the target functions that cannot be ported to a host PC or modifies the unit functions of the target embedded software so that the unit functions can be compiled and executed at the host PC.

1) Removal of Unportable Functions: First, CONBOL Trim identifies functions that cannot be ported to a host PC. Then, these functions are replaced with the corresponding symbolic stub functions that return unconstrained symbolic values. Main causes that make a function unable to run on a host PC are *inline assembly code*, *hardware-dependent code* such as dereference of absolute memory address, and *extensions of RVCT (RealView Compilation Tools) [29]* that are not compatible with GCC.

- *Inline assembly code:*
  Embedded programs often contain inline assembly code to control the target embedded hardware directly. The target functions that contain inline ARM assembly code (the project S runs on the ARM hardware) are removed, since they cannot run on a host PC of x86 architecture.
- *Hardware dependent code:*
  Embedded software often uses memory-mapped I/Os that map hardware control registers to the absolute memory addresses. The target functions that contain memory-mapped I/O code are removed, since they cannot run on a host PC correctly (memory mapped I/Os do not work on different hardware configurations).
- *RVCT compiler extensions:*
  RVCT compiler allows various extensions in target C code to produce optimized executable binary files for target hardware. Some RVCT extensions can be ignored or can be translated to corresponding GCC extensions. If RVCT extensions in a function cannot be translated to GCC compatible code, CONBOL removes the function.

2) Translation of Target Functions: The project S is developed for the ARM architecture and uses RVCT as a compiler. A problem is that RVCT is not fully compatible with GCC on an x86 host PC. In addition, CIL (C Intermediate Language) [21] which is an instrumentation tool that is used by CONBOL Instrumentor does not support RVCT extensions nor GCC-incompatible syntax. Thus, CONBOL modifies the target unit code to be compatible with GCC and CIL as follows:

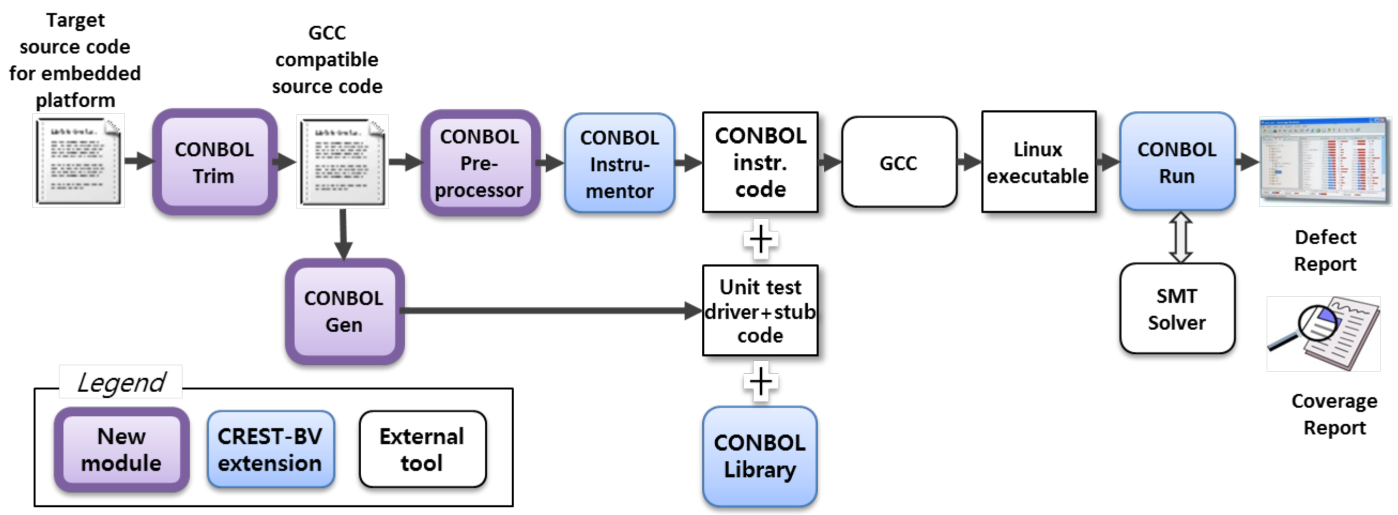

Fig. 1. Overview of the CONBOL framework

- *Translation of the RVCT compiler extensions:*
  If RVCT extensions declare properties of a function but do not impact a target function semantics, CONBOL Trim simply removes the extensions. If RVCT extensions have corresponding GCC extensions, we translate the RVCT extensions to the corresponding GCC extensions. For example, `__align(8)` extension for RVCT (which aligns a data structure in 8 bytes) can be translated to an equivalent GCC extension `__attribute__((aligned(8))`.
- *Resolving type inconsistency:*
  RVCT does not check type strongly between function declaration and corresponding function definition, if they are in separate files (this situation occurs frequently in large software such as the project S). For example, RVCT allows type inconsistency between the types of parameters in function declaration and the types of parameters in function definition, if the declaration and the definition are in different files. CONBOL Trim modifies the function definitions to be type-consistent with the corresponding function declaration and reports this modification to a user.

C. CONBOL Gen: Automated Generation of Unit Test Drivers and Stubs

The CONBOL Gen component automatically generates unit test drivers that specify symbolic input variables for target units and generates stub functions for the functions removed by CONBOL Trim. A generated unit test driver specifies all parameters of the target unit and all global variables that are used by the target unit as symbolic inputs. CONBOL Gen specifies a symbolic input for each variable according to its type as follows:

- *Primitive integer types:*
  If a given parameter variable `x` or a given global variable `y` is a primitive integer type such as `int` and `char`, CONBOL specifies the variable as a symbolic input by using the CONBOL declaration functions such as `CONBOL_int(x)`, `CONBOL_char(y)`, etc. CONBOL does not support floating point symbolic variables, since most SMT solvers do not fully support floating point arithmetics.
- *Array types:*
  If a given variable is an array, CONBOL specifies each array element as a symbolic variable according to the type of the array element. To avoid performance degradation due to too many symbolic variables, a user can specify an upper bound n such that CONBOL specifies only the first n elements of an array as symbolic variables.
- *Structure types:*
  If a given variable `s` is a structure type, CONBOL specifies every primitive field variable of `s` as a symbolic variable recursively (i.e., if `s` contains a structure `t`, the primitive field variables of `t` are declared symbolically). To reduce the complexity of concolic testing, the pointer variables of a structure are not declared symbolically and these pointer variables are assigned with `NULL`.
- *Pointer types:*
  If a given variable `pt` is a pointer to a variable of a type `T`, CONBOL allocates memory space whose size is equal to the size of `T` and assigns the address of the allocated memory to `pt` (i.e., `pt = malloc(sizeof(T))`). If `T` is a primitive type, CONBOL declares the allocated memory as symbolic by using `CONBOL_int()`, etc. If `T` is *not* a primitive type (i.e., a structure), the allocated memory space is declared as symbolic, following the way to specify variables of ‘Structure types’ symbolically.

Figure 2 shows an example of unit test driver code generated by CONBOL Gen. `Node` (lines 1-4) is a structure type that represents a linked list node which contains a character value. The target unit function is `add_last(v)` (lines 7-10) which takes a character `v` as an input and adds a new node containing `v` to the end of the global linked list. `add_last()` uses a global

```
01:typedef struct Node_{
02:    char c;
03:    struct Node_ *next;
04:} Node;
05:Node *head;
06:// Target unit-under-test
07:void add_last(char v){
08:    // add a new node containing v
09:    // to the end of the linked list
10:   ...}
11:// Test driver for the target unit
12:void test_add_last(){
13:    char v1;
14:    head = malloc(sizeof(Node));
15:    CONBOL_char(head->c);
16:    head->next = NULL;
17:    CONBOL_char(v1);
18:    add_last(v1); }
```

Fig. 2. An example of an automatically generated unit test driver

pointer variable `head` (line 5). `test_add_last()`(lines 12-18) is test driver code for `add_last()`.

CONBOL Gen sets all global variables used by the target unit as symbolic inputs. Since the target unit `add_last()` uses `head`, the driver allocates memory space to `head` (line 14). Next, the driver declares all fields of the structure pointed by `head` (except pointer variables) as symbolic variables. In other words, the driver sets `head->c` as a symbolic character variable (line 15) and `head->next` as `NULL`, because `head->next` is a pointer variable of the structure. After the driver finishes setting symbolic global variables, the driver declares function parameters symbolically (line 17). After the driver finishes symbolic input setting, it invokes the target unit function with the symbolic parameters (line 18).

This way of declaring symbolic inputs may declare many symbolic variables. However, a large number of declared symbolic variables does *not* necessarily generate complex symbolic path formulas, because each execution of a target program often accesses only a small subset of symbolic variables of large data structure [18]. For example, in the case study on the project S, each execution of the target unit generates a symbolic path formula that includes less than 14 symbolic variables on average.

In addition, CONBOL generates symbolic stubs for sub-functions called by a target function. These symbolic stub functions simply return symbolic values according to their return types without considering global variable updates. The symbolic return values are constructed in the same way to construct symbolic inputs. Finally, CONBOL replaces the sub-functions of a target function with the symbolic stub functions.

III. Heuristics to Improve the Effectiveness and Precision of Bug Detection

To improve the effectiveness and precision of bug detection, CONBOL utilizes the following heuristics which are implemented in CONBOL Pre-processor (PP):

- *To detect more bugs (see Section III-A)*:
  CONBOL PP inserts `assert(`$expr$`)` to detect more bugs automatically, where $expr$ is a condition to satisfy for correct execution (e.g., `pt != NULL`). Inserted `assert(`$expr$`)` can increase a chance of detecting bugs that violate $expr$, since concolic testing generates input values to make each branching expression (i.e., $expr$) true and false.
- *To reduce false alarms (see Sections III-B– III-D)*:
  First, since the imprecise driver code that violates necessary precondition of a target function can cause false alarms [2], CONBOL PP tries to guarantee a precondition $expr'$ of a target unit by inserting `CONBOL_assume(`$expr'$`)`, which enforces symbolic values to satisfy $expr'$. Second, CONBOL scores every violated assertion and reports only ones with high scores. Finally, after a developer filters out a false alarm, CONBOL inserts an *annotation* at the false alarm location to avoid the same false alarm in later executions.

A. *Inserting* `assert()` *Statements*

CONBOL PP automatically inserts `assert()` statements to detect the following run-time crash bugs. Since a main goal of CONBOL is fully automated testing in a scalable way, CONBOL targets run-time crash bugs which do not require human developers to specify properties to check.

- *Out-of-bound memory access bugs (OOB)*:
  CONBOL inserts `assert(0<=`idx_expr `&&` idx_expr`<size)` right before the statements that contain array read/write operations, where `size` is obtained from the corresponding array declaration statement in the target code. Note that such assertion can increase the probability of detecting an out-of-bound bug, because CONBOL tries to generate test inputs that make idx_expr become negative or greater than the upper bound to explore a false branch of the assertion.
- *Divide-by-zero bugs (DBZ)*:
  CONBOL inserts `assert(` $denominator$`!=0)` right before the statements containing division operators whose denominators are not constants. Similar to the out-of-bound memory assertions, this assertion can increase the probability of detecting a divide-by-zero bug by enforcing CONBOL to generate test inputs that make $denominator$ zero to exercise a false branch of the assertion.
- *Null-pointer-dereference bugs (NPD)*:
  CONBOL inserts `assert(`$pointer$`!=NULL)` right before statements that contain pointer dereference operations. This NPD assertion does not increase a chance to detect a

[2]For example, if an unsorted array is given to a binary search function, the function may cause an error, even when the function is correctly implemented.

```
01:int array[10];
02:void get_ith_element(int i){
03:  return array[i];
04:}
05:// Test driver for get_ith_element()
06:void test_get_ith_element(){
07:  int i, idx;
08:  for(i=0; i<10; i++){
09:    CONBOL_int(array[i]);
10:  }
11:  CONBOL_int(idx);
12:  //CONBOL_assume(0<=idx && idx<10);
13:  get_ith_element(idx);
14:}
```

Fig. 3. Test driver with preconditions

NPD bug, since CONBOL does not analyze pointer variables symbolically. Now CONBOL inserts NPD assertions for information gathering purpose, but we plan to improve CONBOL to analyze pointer variables symbolically in near future.

B. Inserting Constraints to Satisfy Preconditions

CONBOL may generate false alarms due to the imprecise unit test driver that violates *preconditions* of a target unit under test. Figure 3 shows an example of such false alarm. The target unit `get_ith_element()`(lines 2-4) receives an index to an element of `array` declared at line 1 and returns the element of `array` at the index. The test driver `test_get_ith_element()` sets all elements of `array` as symbolic variables (lines 8–10) and `idx` as a symbolic variable (line 11), and executes `get_ith_element(idx)` (line 13). Note that `test_get_ith_element()` (lines 6–14) can crash `get_ith_element()` due to the out-of-bound array access, since `idx` is declared as a symbolic variable and it can be larger than the array size. However, this violation can be a false alarm, if `get_ith_element(idx)` is always invoked with `idx` between 0 and 9 in the target program.

Developers often write a unit function based on the assumption that the unit will be called with 'valid' parameters. Thus, to reduce false alarms, it is important to insert constraints to satisfy such preconditions of a target unit. [3] CONBOL PP inserts such constraints of target functions automatically by using `CONBOL_assume(expr)`. [4] In the above example where a precondition of `get_ith_element(idx)` is `0<=idx && idx<10`, the unit test driver should generate symbolic input values that satisfy the precondition, which is enforced by `CONBOL_assume(0<=idx && idx<10)` at line 12.

Currently, CONBOL PP inserts the following three types of constraints:

- *Preconditions for array indexes*:
  To avoid false alarms due to infeasible out-of-array indexes, CONBOL PP inserts `CONBOL_assume(0<=` idx_expr && idx_expr `<size)` before the invocation of a target function as a precondition to satisfy for array accesses through idx_expr in the target function, where `size` is obtained from the corresponding array declaration statement in the target code. However, to keep the chance of detecting array out-of-index bugs, CONBOL PP inserts constraints on idx_expr *only if* idx_expr satisfies all of the following conditions:
  1) idx_expr should be a form of `x + a` where `x` is a symbolic integer variable and `a` is an integer constant (i.e., `a[x-1] = ... `).
  2) `x` should *not* be updated in the target function.
  3) The target function should *not* check the value of `x` (e.g., `if(x<=10+y)`...).
- *Preconditions for constant parameters*:
  Developers often write a function whose parameter should have one of the pre-defined constant values. For example, the third parameter of `fseek()` C standard function should be one of the three constant values `SEEK_SET`, `SEEK_CUR` and `SEEK_END`. Any values other than these three constants are invalid values and can cause false alarms when `fseek()` is tested. Thus, CONBOL PP inserts constraints to generate a symbolic value that is one of the valid constant parameters for a target unit.
  CONBOL PP identifies such a function `f()` whose parameter should have one of predefined constant values by looking at the function invocation statements. If all statements that invoke `f()` in the target code pass a constant as a parameter of `f()`, CONBOL PP inserts constraints to generate only such constant values for the parameter.
- *Preconditions for* `enum` *values*:
  When an `enum` variable is declared symbolically, this variable is declared as a symbolic integer variable. To prevent false alarms due to undefined `enum` values, CONBOL PP inserts constraints to generate only integer values defined in the corresponding `enum` type for an `enum` variable.

C. Scoring of Alarms

To reduce false alarms, CONBOL assigns a score to each violated assertion that CONBOL inserts (see Section III-A) and reports only violated assertions with scores larger than a threshold. Main scoring rules for violated assertions are as follows and CONBOL reports only violated assertions whose scores are six or higher: [5]

[3]Constraints to satisfy such preconditions may cause false negatives, if a developer has incorrect assumption. However, utilizing the constraints to reduce false alarms even at the cost of false negatives can be a good strategy, because it can be more important to reduce false alarms than to reduce false negatives in industry targeting smartphone market (see Section VI-B).

[4]`CONBOL_assume(expr)` is a macro of `if(!expr) exit(0);`. If a current test case tc violates a given precondition (i.e., `expr` becomes false), CONBOL immediately terminates a target unit execution with tc and removes tc, since tc can raise a false alarm.

[5]CONBOL has 13 scoring rules based on the target code and runtime execution information. The other 10 scoring rules were not effective to filter out false alarms, and not applied in this case study.

1) Every violated assertion gets 5 as a default score.
2) For each violated assertion which contains a variable `x`, if the target function containing the assertion checks the value of `x` (e.g., `if(x < y+1)`...), the score of the assertion increases by 1. A rationale for this rule is that an explicit check of `x` in the target function indicates that the developer of the function considers `x` important and the assertion on `x` is important consequently.
3) For each violated assertion `assert(`*expr*`)`, the score of the assertion decreases by 1, if *expr* appears five or more times in other violated assertions in the entire target software. A rationale for this rule is the assumption that a developer writes code correctly most of time so that target code does not have a same bug that appears many times in different locations of the target program.

D. Annotation Mechanism to Utilize User Feedback

CONBOL PP utilizes a user feedback through annotations in a target code. CONBOL annotation is specified as a comment starting with `/*CBL`. A user can guide CONBOL through this annotation mechanism to reduce false alarms. For example, if a developer identifies a false alarm located at line l, CONBOL inserts the following annotation at line l:
`/*CBL action=suppress,object=none,log=false...`
By using this annotation, the identified false alarms will be suppressed for later executions.

IV. Case Study on Samsung Project S

The goal of this case study is to evaluate the *effectiveness* (in terms of a number of detected bugs) and *efficiency* (in terms of testing time and false alarm ratio) of CONBOL for large-scale industrial embedded software. For this purpose, we have applied CONBOL to four million lines long embedded software developed by Samsung Electronics (calling it project S in this paper).

A. Target Project Description

The project S has been developed for smartphones. The rough statistics on the structure of the project S (written in mainly C) is as follows: [6]

- Total number of directories: 3123
- Total number of source files: 7243
- Total number of header files: 10401
- Total number of functions: 48743
  - Total number of functions having more than one branch: 29324
- Total number of branches: 397854
- Total lines of code: four million lines of C codes

Project S targets ARM platform and uses RVCT compiler infrastructure. We chose the project S as our target program, because it is important software for commercial smartphones. In addition, the project S had suffered subtle bugs, which consumed a large amount of developer time and resource.

[6]To secure the intellectual property rights of Samsung Electronics, detailed information on the project S is not written in this paper.

B. Experimental Setup

Unit testing has several advantages to improve software quality such as early detection of bugs and corner case bug detection [28]. Unit testing has additional benefits for embedded software, since unit testing has less dependency on the target embedded platform by building testing driver/stubs [20]. Thus, we decided to apply CONBOL to the project S in unit level.

CONBOL uses reverse depth-first search strategy [5] to explore execution paths of the target unit and increase branch coverage fast. Unit testing of a target unit terminates when

- An assertion to detect a crash bug is violated, or
- All possible execution paths are explored, or
- All test executions of a target unit spend 30 seconds (Timeout1).

In addition, we enforce Timeout2 by which a single test execution of a target unit terminates when the execution takes 15 seconds.

The experiments were performed on a machine that has Intel i5 3570K (3.4GHz) and with 4GB RAM, running Debian 6.0.4 32bit version.

V. Case Study Results on the Project S

A. Results of CONBOL Trim, CONBOL Gen, and CONBOL PP

CONBOL Trim removed unportable functions of the project S and the number of final target functions is 25425 out of the 29324 functions that have more than one branch. Among 3899 (=29324-25425) removed functions, 2825 functions were removed due to ARM inline assembly, 806 functions were removed due to hardware dependent code, and 268 functions were removed due to RVCT compiler extension (see Section II-B). As a result, 86.7% (=25425/29324) of the target functions were tested by CONBOL.

The size of symbolic setting portions generated by CONBOL Gen is 60.8 lines long on average, declaring 58.9 symbolic variables and containing 9.51 symbolic stub sub-functions on average. CONBOL PP inserted 14.3 assertions in each target function on average (i.e., 8.0 NPD assertions, 6.2 OOB assertions, and 0.1 DBZ assertions on average). CONBOL PP also inserted 2.3 precondition constraints in each target function on average (i.e., 1.4, 0.6, and 0.3 precondition constraints for `enum` variables, array indexes [7], and constant parameters, respectively).

B. Result on Detected Bugs

After testing the 25425 target functions and applying the false alarm reduction techniques, CONBOL reported 277 alarms.

We filtered out false alarms by reviewing the relevant target source code, especially the calling context of the target functions. Interestingly, similar alarms occurred repeatedly so that we could remove most of the alarms without much difficulty. For example, we observed many violations of similar assert conditions in similar context. In addition, many

[7]The average number of inserted precondition constraints for array indexes is small (i.e., 0.6) due to the three strict conditions (Section III-B).

alarms regarding specific variables were false alarms due to imprecise environments. For example, all violations of `assert(gd[i].f!=0)` were false alarms, since all target functions that access `gd` are called only after the initialization function `init_gd()` that assigns all fields of the elements of `gd` correctly (i.e., `init_gd()` assigns `gd[i].f` with non-zero for all possible `i`'s in real executions). Two authors of Samsung without prior knowledge on the project S spent a week to remove false alarms. Finally, we reported 50 alarms to the original developers and the following 24 crash bugs among these 50 alarms were confirmed by the original developers of the project S.

- *13 array out-of-index bugs*:
  More than a half of detected bugs are OOB bugs, since the project S utilizes complex data structures containing arrays. Eight OOB bugs are made, because the index checking statement (line 4) is located after the array access (line 3) as shown below:
  ```
  1:void foo(u8 index) {
  2:  ...
  3:  g[index-1] = ...;
  4:  if((index==0)||(index>10)) return;
  5:  ...}
  ```
  Note that CONBOL does not insert a precondition constraint for such code, since the code checks the range of `index` at line 4 (see Section III-B). The other bugs are due to incomplete checking of the values of index variables.
- *6 divide-by-zero bugs*:
  Two of the detected divide-by-zero bugs are as following:
  ```
  1:u32 foo(u32 t, ...) {
  2:  ...
  3:  if (t != 0) { ...
  4:    if(size < 2) {z=z/(t/10);}
  5:    else if (size < 3) {z=z/(t/100);}
  6:    else z=z/(t/1000);}
  7:  ...  }
  ```
  Lines 4–6 will raise divide-by-zero errors in spite of line 3 that checks a value of the denominator `t` due to integer division, if an unsigned 32-bit integer `t` is less than 10, 100, and 1000, respectively. The other four bugs are due to missing tests to check if denominator values are zero.
- *5 null-pointer dereference bugs*:
  Although CONBOL does not support symbolic pointer varaibles, CONBOL detected five null-pointer dereference bugs, because symbolic test drivers generated by CONBOL set a pointer variable in a structure as `NULL` (Section II-C).

C. Coverage and Time Costs

Table I describes a number of generated test cases, branch coverage, and time cost of CONBOL for the 25425 target functions. CONBOL covered 59.6% of the target branches with 0.8 million test cases in 15.8 hours. After removing the time cost caused by the target functions that reached timeout1 or timeout2, CONBOL spent 9.0 hours.

TABLE I
BRANCH COVERAGES AND TIME COSTS

Total # of test cases generated	0.8×10^6
Branch coverage(%)	59.6
Total time spent (hour)	15.8
# of functions that reached timeout1	742 (TO:30s)
# of functions that reached timeout2	134 (TO:15s)
Time cost w/o timeout (hour)	9.0

TABLE II
EFFECTIVENESS OF FALSE ALARM REDUCTION TECHNIQUES

# of reported alarms	OOB	NPD	DBZ	Sum
Total #	3235	2588	61	5884
W/ precondition constraints	2486	2511	58	5055
W/ scoring rules	220	42	15	277

Regarding the branch coverage result, we found the following reasons for the uncovered 40.4% (=100%-59.6%) of the branches. [8] First, a test execution terminated at an assertion violation and no further branches were covered in the execution (see Section IV-B), which makes 10.8% of the target branches uncovered. Second, functions that reach timeout were not covered completely, which makes 9.3% of the branches uncovered. [9] The remaining 20.3% of the branches were not covered due to the limitations of CONBOL such as lack of symbolic pointer support, setting pointers as `NULL` in a symbolic `struct` input, no support for floating pointer arithmetic, external libraries, etc. [10]

D. Effectiveness of the False Alarm Reduction Techniques

Without applying any false alarm reduction techniques (i.e., without precondition constraints, nor scoring rules), CONBOL generated 5884 (=3235+2588+61) alarms (the second row of Table II).

By inserting the precondition constraints (Section III-B), 14.1% (=$\frac{5884-5055}{5884}$) of alarms were removed (see the third row of the table). Note that OOB alarms were reduced 23.2% (=$\frac{3235-2486}{3235}$) on average, respectively mainly due to the precondition constraints for array indexes.

Finally, after applying the scoring rules (Section III-C), 94.5% (=$\frac{5055-277}{5055}$) of the alarms were removed (the fourth row of the table). For example, 54.3% of the alarms have score 4 due to the rules 1) and 3), 36.8% of the alarms have score 5 due to the rules 1), 2) and 3) together, and 3.5% of the alarms have score 5 due to the rule 1) only.

[8]CONBOL Instrumentor transforms a target program to an equivalent extended version whose branches contain only one atomic condition per branch. Thus, the branch coverage achieved on the original program is much higher than the coverage data in Table I on the extended target program.

[9]In the exploratory experiments, increased timeouts did not improve the coverage much.

[10]10.8% and 9.3% were calculated by simply counting the uncovered branches of the target units which raised an alarm or reached timeout (Section IV-B). Thus, we think that more than 20.3 % of the target branches were uncovered due to the limitations of CONBOL.

VI. Lessons Learned

A. *Effectiveness and Efficiency of the CONBOL Framework*

Through the case study on the project S, we could confirm that CONBOL is an effective framework by detecting 24 crash bugs in large embedded software (see Section V-B). A main reason why these 24 bugs had not been detected by the developers is that the bugs can be triggered only in corner-case scenarios, which are hard to imagine by human engineers. Concolic testing technique, as demonstrated in the other industrial case studies [14], [15], [18], have strengths in exploring corner-case scenarios and detect subtle bugs. In addition, CONBOL spent less than one day to detect those crash bugs, which is affordable time cost in industrial setting. To our knowledge, this case study is the first industrial case study that demonstrates the practicality of concolic testing technique on multi million lines long industrial embedded software.

Another interesting observation is that the project S had been regularly checked by using static analysis tools such as Coverity Prevent. [11] These 24 crash bugs reported by CONBOL had not been detected by those static analysis tools, since static analysis tools often perform only simple analysis for fast bug detection and report alarms only when the scores of the alarms are higher than some threshold to reduce false alarms. Although both static analysis tools and CONBOL target crash bugs, it is a good idea to apply static analysis tools and CONBOL together, since they can complement each other.

B. *Issues for Successful Technology Transfer to Industry*

The development team of the project S decided to incorporate CONBOL in their development process after a series of discussions with us regarding the needs of the developers and the detailed information of CONBOL. Main issues of the discussions are as follows;

1) Trade-off between Software Quality and Time-to-Market: We found that the developers were not concerned much for the detected crash bugs, since most of the bugs cause errors in corner-case scenarios (i.e., they think that a user would experience the corresponding errors very rarely in daily use), but concerned for extra overhead caused by adopting CONBOL (i.e., steep learning curve, manual steps required to apply CONBOL, long testing time, reviewing many alarms, etc.), since time-to-market is critical for a smartphone market.

We understood the viewpoint of the developers for a smartphone market and responded as follows. First, we had designed CONBOL to satisfy such viewpoint of the developers by increasing the degree of automation of unit testing by generating driver/stubs automatically (Section II-C) and reducing false alarms (Sections III-B–III-D). Second, regarding the concern of steep learning curve, we made seminars to help the developer understand CONBOL and concolic testing. Third, we made it clear that the execution time of CONBOL could be reduced significantly by distributing target units to multiple testing machines, since CONBOL tests every unit independently. Finally, we explained that reviewing of alarms might not consume a large amount of time, since many of the detected alarms showed similar patterns and, thus, many alarms could be removed together. Furthermore, this manual effort to review the alarms is one-time cost for the initial version of target software. For subsequent revisions of the target software, only a small number of new alarms will be reported, since the false alarms detected in the previous version are suppressed by the annotation (Section III-D).

2) Delivering Confidence in the Advanced Testing Techniques: Although CONBOL detected new crash bugs, the developers were uncertain about the benefit of using CONBOL due to their negative preconception on automated testing tools (most developers had tried and found several automated testing tools working for demo cases, but not really working for their previous projects). In other words, they did not understand the underlying concolic testing techniques and, thus, thought CONBOL as just yet-another automated testing tool based on ad-hoc heuristics.

To give confidence in the benefit of using CONBOL, we delivered a detailed report on the CONBOL experiment on the project S. In addition, we made a series of seminars to help the developers understand both strengths and limitations of concolic testing techniques. As a result, the developers started to admit that CONBOL can generate test cases systematically and at modest cost to detect bugs that are hard to find by using random testing or manual testing.

3) Improving Tool Maturity of CONBOL: Initially, the developers concerned about hidden manual cost to apply CONBOL to their project, such as modification of target source building scripts, lack of GUI that costs more labor to analyze bug reports, lack of integration with existing test case management tools, etc.

We persuaded the developers that most of these issues are implementation issues and can be solved so that CONBOL can be grown to a fully usable automated tool by investing tool implementation effort and satisfying the needs of the developers. This is possible because CONBOL was an in-house tool developed by the Samsung engineers.

C. *Technical Challenges for Concolic Unit Testing*

Although the case study on the project S was successful, we observed the following technical challenges to solve:

- *Support for Complex Symbolic Data Structure*:
  As shown in Section V-C, one of the main reasons for low branch coverage was lack of support for complex symbolic data structure. Large industrial software often utilizes complex recursive data structure and bugs are frequently introduced to routines that handle the data structure. Thus, it is necessary to analyze complex data structure symbolically, which requires support for symbolic pointer arithmetic and memory operations as well as efficient algorithm to generate complex data structure [22].

[11] The original developers said that Coverity Prevent had reported less than 100 alarms on average. Five engineers spent one day to review these alarms on average.

- *Necessity of Branch-oriented Search Strategy*:
  Another reason for the low branch coverage was due to the timeout problems, which are often caused by a large number of test executions due to symbolic variable dependent loops. Although several search strategies (pathcrawler [36], Godefroid et al. [13], Saxena et al. [31]) were proposed to achieve high branch coverage fast by handling loop efficiently, they often fail to achieve the goal in practice. As branch coverage is a standard metric to measure quality of testing in industry, improving branch-oriented search strategy for concolic testing can be an important research direction.
- *Automated Generation of Effective and Efficient Unit Test Driver/Stubs*:
  The current method of CONBOL to build test driver/stubs is simple and can be refined further to improve precision of unit testing and to save testing cost at the same time. For example, a unit testing driver can include sub-functions that are 'closely' related to the target function and have low 'cost' to include. It will be an interesting research direction to define proper 'closeness' and 'cost' metrics for effective and efficient concolic unit testing.
- *Tool Assistance for Test Oracle Specification:*
  Although CONBOL detected dozens of crash bugs, the project S may still have functional bugs. The developers of the project S recognized the importance of user-specified functional assertions, but they were too busy to specify meaningful functional assertions. Although we can utilize automated invariant generation techniques to specify test oracles, they are not mature enough to generate meaningful assertions for large industrial software. Instead, as the first step toward (semi) automated test oracle generation, it will be more practical to develop a tool to support a developer to specify assertions fast and conveniently by providing inferred information on program segment visually.

VII. Related Work

A. Automated Unit Testing Techniques

There has been active research on automated unit test techniques. Among them, the simplest but most popular technique is *random testing* [7], [23], [24]. Random testing generates a large number of concrete test cases by using random values in short time. Random testing, however, often fail to achieve high coverage even with a large amount of test cases.

Other techniques generate test cases based on the systematic reasoning of a target source code. Beyer et al. [4] use predicate abstraction-based model checking, and Visser et al. [35] use explicit-state model checking to generate concrete test cases based on the generated counter examples. Csallner et al. [8] use static analysis to detect candidate bugs and generate concrete test cases to invoke the bugs actually. A drawback of these systematic techniques is low accuracy caused by the incomplete reasoning engines such as underlying constraint solvers when applied to complex real-world programs. For example, predicate abstraction-based model checking [4] does not handle program code that uses complex data structure that uses complex pointer arithmetic [16].

In addition, it is an important issue to how to build environment/driver/stub, as shown in this paper. Thummalapenta et al. [33] describes a systematic guideline to build a symbolic environment model from an existing manual unit testing environment. Zhang et al. [19] describes how to model a symbolic environment for cloud applications to generate test case automatically by using Pex [34], where they achieved 76.8% block coverage of PhluffyFotos (photo gallery software).

B. Concolic Testing Tools

Since DART [11] and CUTE [32] were developed in 2005, various concolic testing tools [5], [34], [6], [18], [17] have been developed, Among these concolic testing tools, CUTE [32] and PEX [34] explicitly claim that they target unit tests. Garg et al. [26] extends RANDOOP [24] which is a feedback-guided automated testing tool by adopting concolic testing techniques for C/C++ programs. CONBOL is a concolic testing framework specialized to unit test *embedded software* in a sense that it provides several functionalities which are not supported by other concolic testing tools such as porting of unit source codes (CONBOL Trim), unit test driver generation (CONBOL Gen), and improved bug detection and false alarm reduction (CONBOL PP).

C. Concolic Testing of Embedded Software

Concolic testing has been applied to various application domains such as sensor networks [30], web applications [1], database applications [10], [25], etc. However, only few case studies have been conducted to apply concolic testing to embedded software due to the challenges aforementioned in Section I. For example, Kim et al. applied an open-source concolic testing tool CREST to flash memory device driver code [14] and mobile phone software [15], where the target programs were relatively easy to port to a host PC, which is very different from the application of the CONBOL framework to the project S reported in this paper. As an another example, Bardin and Herrmann [2], [3] applied concolic testing to embedded software running on microprocessors. They translated machine code into intermediate representation to apply concolic testing. They applied this approach to string functions in C library and unit functions of industrial embedded software such as aircraft engined power controller. This approach is not suitable for the project S, because we have to develop a tool for machine code analysis, which will cause unacceptable development overhead in industrial setting.

VIII. Conclusion and Future Work

We have presented the CONBOL testing framework to unit test large size embedded software automatically. CONBOL tests target units on a host PC platform by generating symbolic unit testing drivers/stubs and test cases to explore a large number of possible execution paths systematically. In addition, CONBOL utilizes several heuristics to reduce false alarms caused by the imprecise drivers. We have demonstrated that CONBOL is a practically effective testing framework through the case study on four million lines long industrial embedded software. In this study, CONBOL detected 24 new crash bugs that had not been detected by manual testing nor by static analysis tools. We plan to refine CONBOL to support symbolic pointer variables and to generate a more accurate unit testing driver by inserting constraints based on clustering of false alarm feedbacks by developers through data mining techniques. Finally, we will apply the CONBOL framework to other large scale industrial embedded projects to evaluate the advantages and weaknesses of the framework in more detail.

Acknowledgment

This work was supported by the DMC department of Samsung Electronics, Mid-career Researcher Program through NRF grant funded by MSIP (No. 2012046172), the IT R&D program of MKE/KEIT [10041752, Research and Development of Dual Operating System Architecture with High-Reliable RTOS and High-Performance OS], and the MSIP, Korea in the ICT R&D Program 2013.

References

[1] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst. Finding bugs in dynamic web applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[2] S. Bardin and P. Herrmann. Structural testing of executables. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2008.

[3] S. Bardin and P. Herrmann. Osmose: automatic structural testing of executables. *Journal of Software Testing, Verification, and Reliability (STVR)*, 21(1):29–54, 2011.

[4] D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating test from counterexamples. In *International Conference on Software Engineering (ICSE)*, 2004.

[5] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, 2008.

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

[7] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software-Practice and Experience (SPE)*, 34(11):1025–1050, Sept 2004.

[8] C. Csallner and Y. Smaragdakis. Check 'n' crash: Combining static checking and testing. In *International Conference on Software Engineering (ICSE)*, 2005.

[9] B. Elkarablieh, R. Godefroid, and M. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA*, 2009.

[10] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation (PLDI)*, 2005.

[12] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[13] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[14] 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.

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

[16] M. Kim, Y. Kim, and H. Kim. Comparative study on software model checkers as unit testing tools: An industrial case study. *IEEE Transactions on Software Engineering (TSE)*, 37(2):146–160, March 2011.

[17] M. Kim, Y. Kim, and G. Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[18] 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 *International Conference on Software Engineering (ICSE)*, 2012. SEiP track.

[19] L.Zhang, T.Xie, N.Tillmann, P.Halleux, X.Ma, and J.Lv. Environment modeling for automated testing of cloud applications. *IEEE Software*, 29, 2012.

[20] M.Kucharski. Making unit testing practical for embedded development. In *Electronic Design*, Nov 2011. http://electronicdesign.com/article/embedded/Making-Unit-Testing-Practical-for-Embedded-Development.

[21] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction (CC)*, 2002.

[22] R. Nokhbeh Zaeem and S. Khurshid. Test input generation using dynamic programming. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 34:1–34:11, New York, NY, USA, 2012. ACM.

[23] C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[24] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE)*, 2007.

[25] K. Pan, X. Wu, and T. Xie. Generating program inputs for database application testing. In *Automated Software Engineering (ASE)*, 2011.

[26] P.Garg, F.Ivancic, G.Balakrishnan, N.Maeda, and A.Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *International Conference on Software Engineering (ICSE)*, 2013.

[27] P.Runeson. A survey of unit testing practices. *IEEE Software*, July/Aug 2006.

[28] R.Osherove. *The Art of Unit Testing*. Manning Publications, 2009.

[29] Realview compilation tools. http://www.arm.com/products/tools/software-tools/rvds/arm-compiler.php.

[30] R. Sasnauskas, O. Landsiedel, M. h. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *International Conference on Information Processing in Sensor Networks (IPSN)*, 2010.

[31] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

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

[33] S.Thummalapenta, M.Marri, T.Xie, N.Tillmann, and J.Halleux. Retrofitting unit tests for parameterized unit testing. 2011.

[34] N. Tillmann and W. Schulte. Parameterized unit tests. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2005.

[35] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[36] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, 2005.