Concolic Testing for High Test Coverage and Reduced Human Effort in Automotive Industry

Yunho Kim
School of Computing
KAIST
Daejeon, South Korea
yunho.kim03@gmail.com

Dongju Lee
Software Verification Team
Hyundai Mobis
Yongin, South Korea
dongju.lee@mobis.co.kr

Junki Baek
Software Verification Team
Hyundai Mobis
Yongin, South Korea
jk.baek@mobis.co.kr

Moonzoo Kim
School of Computing
KAIST
Daejeon, South Korea
moonzoo@cs.kaist.ac.kr

***Abstract*—The importance of automotive software has been rapidly increasing because software now controls many components in motor vehicles such as window controller, smart-key system, and tire pressure monitoring system. Consequently, the automotive industry spends a large amount of human effort testing automotive software and is interested in automated software testing techniques that can ensure high-quality automotive software with reduced human effort.**

In this paper, we report our industrial experience applying concolic testing to automotive software developed by Hyundai Mobis. We have developed an automated testing framework MAIST that automatically generates the test driver, stubs, and test inputs to a target task by applying concolic testing. As a result, MAIST has achieved 90.5% branch coverage and 77.8% MC/DC coverage on the integrated body unit (IBU) software. Furthermore, it reduced the cost of IBU coverage testing by reducing the manual testing effort for coverage testing by 53.3%.

***Keywords*-Automated test generation, concolic testing, automotive software, coverage testing**

I. Introduction

The automotive industry has developed automotive software to control various components in the motor vehicle, for example, the body control module (BCM), smart-key system (SMK), and tire pressure monitoring system (TPMS) [1], [2]. As automotive software becomes larger and more complex with the addition of newly introduced automated features (e.g., advanced driver assistance systems) and more sophisticated functionality (e.g., driving mode systems) [3], [4], the cost of testing automotive software is rapidly increasing. Also, it is difficult for human engineers to develop test inputs that can ensure high-quality automotive software within tight software development schedules and budgets. To resolve these problems, the automotive industry is trying to apply automated software testing techniques to achieve high code coverage with reduced human effort.

Concolic testing [5] (a.k.a. dynamic symbolic execution [6]) has been applied to automatically generate test inputs for software in various industries. Concolic testing combines dynamic concrete execution and static symbolic execution to explore all possible execution paths of a target program, which can achieve high code coverage. Concolic testing has been applied to various industrial projects (e.g., flash memory device driver [7], mobile phone software [8], [9], and large-scale embedded software [10]) and has effectively improved the quality of industrial software by increasing test coverage and detecting corner-case bugs with modest human effort.

While we were working to apply concolic testing to automotive software developed by Mobis, we observed the following technical challenges to resolve to successfully apply automated test generation techniques:

1) We need to generate test drivers and stubs carefully to achieve high unit test coverage while avoiding generating test cases corresponding to the executions that are not feasible at the system-level. Otherwise (e.g., generating naive test drivers and stubs that provide unconstrained symbolic inputs to every function in a target program), we will waste human effort to manually filter out infeasible tests that lead to misleading high coverage and false alarms.
2) Current concolic testing tools do not support symbolic bit-fields in C which are frequently used for automotive software.[1] For example, automotive software uses bit-fields in message packets in the controller area network (CAN) bus to save memory and bus bandwidth. However, most concolic testing tools do not support symbolic bit-fields since a bit-field does not have a memory address (Sect. III-D) and most programs running on PCs rarely use bit-fields.
3) Although automotive software uses function pointers to simplify code to dynamically select a function to execute, concolic testing techniques and tools do not support symbolic function pointers due to the limitation of SMT (Satisfiability Modulo Theories) solvers.

To address the above challenges, we have developed an automated testing framework MAIST. MAIST automatically generates the test driver, stubs, and test inputs for a target unit using concolic testing. MAIST achieved 90.5% branch coverage and 77.8% MC/DC coverage of the IBU software (Sect. V-A). Also, MAIST reduced the manual testing effort by 53.3% for coverage testing of IBU (Sect. V-C). [2]

The main contributions of this paper are as follows:

[1] A bit-field `x:m` is an integer variable in a `struct` variable which has only `m` bits. For example, `unsigned int x:3` can represent only 0 to 7.

[2] Several newspapers reported these successful results [11]–[13].

1) We have developed MAIST which automatically generates the test driver, stubs, and test inputs achieving high coverage for automotive software (MAIST generates test input by using a concolic testing tool CROWN [14]).
2) We have identified the technical challenges in applications of concolic testing to automotive software and describe how MAIST resolves them (i.e., *task*-oriented driver/stub generation (Sect. III-C1), symbolic bit-field supports (Sect. III-D), and symbolic setting for function pointers (Sect. III-C3)). Thus, this paper can support field engineers in the automotive industry to adopt automated test generation with less trial-and-error.
3) To the authors' best knowledge, this is the first industrial study that concretely demonstrates reduced human effort (i.e., human effort reduced by 53.3%) by applying concolic testing in the automotive industry (Sect. V-C). Thus, this study can promote the adoption of concolic testing in the automotive industry.
4) This paper shares lessons learned and valuable information for both field engineers in the automotive industry and researchers who develop automated testing techniques (Sect. VI).

The rest of the paper is organized as follows. Sect. II explains the target project (i.e., IBU). Sect. III describes the MAIST framework. Sect. IV explains how we have applied MAIST to IBU. Sect. V describes the experiment results. Sect. VI presents lessons learned from this industrial study. Sect. VII discusses related work. Finally, Sect. VIII concludes this paper with future work.

II. Target Project: Controller Software for Integrated Body Unit

A. Overview

Integrated body unit (IBU) is the first AUTOSAR-compliant electronic control unit (ECU) developed by Mobis. IBU integrates the three ECUs (i.e., body control module (BCM), smart key system (SMK), and tire pressure monitoring system (TPMS)) into one ECU to reduce the physical unit size and production cost. Mobis has developed more than 10 different versions of IBUs targeting various motor vehicle models.

We chose the IBU software as our target project because IBU is essential to drive motor vehicles safely. For example, SMK has an automotive safety integrity level (ASIL) of B and the handle controller in SMK has ASIL D. Mobis spends a considerable amount of test engineer resource to test the IBU software (from now on, IBU software will be referred to as IBU).

B. Target Project Statistics

Table I shows that IBU consists of total 254 source files and 3,479 functions having 17,858 branches. Maximum and average cyclomatic complexity of the functions are 24 and 4.9, respectively. IBU consists of *tasks* which are mostly minimal independent units. A task t consists of

1) an entry function t_e which is defined as a non-static function in a target source file s, and
2) the callee functions that are directly or transitively invoked by the entry function t_e and defined in the same source file s.

IBU has a total of 767 tasks (i.e., 767 non-static functions as their entry functions). Each source file contains 3.0 tasks (=767/254) on average. Each task consists of 6.3 functions on average.

TABLE I
The code statistics of IBU

Module	#files	#functions			LoC	#branches	Cyclomatic comp.		
		non-static	static	total			min	max	avg
BCM	27	145	511	656	52690	2873	1	15	4.3
SMK	198	554	1967	2521	134877	13768	1	24	5.8
TPMS	29	68	234	302	15951	1217	1	12	4.1
Total	254	767	2712	3479	203518	17858	1	24	4.9

```
01:int rpm_FL,rpm_FR,rpm_RL,rpm_RR;
02:int angle_FL,angle_FR;
03:int press_FL,press_FR,press_RL,press_RR;
04:int mode,dir,speed;
05:void drive_mode_check(){
06:  ...
07:  if (mode==DRIVE && speed > 0 &&
08:  ((dir==LEFT && angle_FL<0 && angle_FR<0 &&
09:  rpm_FL <= rpm_FR && rpm_RL <= rpm_RR) ||
10:  (dir==RIGHT && angle_FL>0 && angle_FR>0 &&
11:  rpm_FL >= rpm_FR && rpm_RL >= rpm_RR) ||
12:  (dir==STRAIGHT && angle_FL==0 && angle_FR==0 &&
13:  rpm_FL==rpm_FR && rpm_RL==rpm_RR)) &&
14:  (L_PRESSURE<press_FL && press_FL<H_PRESSURE) &&
15:  (L_PRESSURE<press_FR && press_FR<H_PRESSURE) &&
16:  (L_PRESSURE<press_RL && press_RL<H_PRESSURE) &&
17:  (L_PRESSURE<press_RR && press_RR<H_PRESSURE))
18:  {...}}
```

Fig. 1. A code example of IBU that reads a large number of input variables and evaluates a complex branch condition

C. Challenges for Manual Testing IBU

Manual derivation of test inputs for IBU has the following obstacles:

1) *A large number of input variables*: Most functions of IBU take a large number of inputs through parameters and global variables because IBU checks a various status of a motor vehicle such as speed, wheel angles, tire pressure, etc.
2) *Complex branch conditions*: The branch conditions of IBU are complex in terms of the number of the logical operators (e.g., `&&`, `||`) used in a branch condition. This is because IBU checks complex conditions on complicated status data obtained from various components of a motor vehicle.

Fig. 1 is a code example that reads a large number of input variables and evaluates a complex branch condition. The function `drive_mode_check` (Lines 5–18) reads a total of 13 input variables (Lines 1–4) to evaluate the branch condition which has 24 logical operators (Lines 7–17).

Also, we compare the number of the input variables and the complexity of the branch conditions of IBU with the five most popular open-source C projects in OpenHub [15]: Apache,

```
01:#define M_MAX 10
02:int model[M_MAX];
03:void f(int x){
04:  ...
05:  g(x % M_MAX);}
06:#define TM9 9
07:static void g(int idx){
08:  ...
09:  for (int i=0; i<idx; i++){
10:    // FALSE ALARM
11:    if (model[i] == TM9){ ... }
12:  ...}}
13:void driver_g(){
14:  int param1;
15:  SYM_int(param1);
16:  g(param1);}
```

Fig. 2. An example of false alarms raised by a naive test driver

MySQL, Subversion, PHP and Bash. [3] Each function of IBU has 52.9% larger number of input variables (i.e., 10.4) than the open-source programs (i.e., 6.8) on average. Similarly, IBU's branch conditions have 2.2 times more number (i.e., 2.8) of `&&` and `||` than the open-source programs (i.e., 1.3).

However, concolic testing can effectively resolve the above challenges for manual testing because concolic testing can automatically generate input values that exercise all combinations of both outcomes of the complex branching conditions one by one.

D. Challenges for Concolic Testing IBU

Achieving high coverage of IBU is still challenging for concolic testing for the following reasons:

1) Infeasible unit test executions generated: Concolic testing may generate an infeasible unit test execution (i.e., a test that is not feasible at system-level) which can report misleading coverage results and waste human engineers' effort to filter out false alarms (e.g., crashes caused by infeasible test executions).

Fig. 2 is a code example that shows that a naive function-oriented test driver raises a false alarm. Suppose that only `f` invokes `g`. `f` calls `g` with an argument `(x % M_MAX)` which is always less than 10 (i.e., Line 11 is always safe since `model` has 10 elements (Lines 1–2)). However, a naive test driver `driver_g` (Lines 13–16) directly calls `g` with a symbolic argument value and raises access out-of-bound alarms (i.e., false alarms) at Line 11 because `idx` can be larger than 10.

2) No support for symbolic bit-fields: IBU uses bit-fields to save memory space and CAN bus bandwidth. The existing concolic testing tools do not support symbolic bit-field [4] and may not achieve high coverage of IBU because they cannot guide symbolic executions to cover branches whose conditions depend on bit-fields.

Fig. 3 shows a code example where concolic testing may not cover the branch at Line 12. This is because the branching condition depends on a bit-field `u.s.b7` (Line 11) which cannot have a symbolic value due to the lack of symbolic bit-field support.

```
01: struct ST{
02:   int b0:1;
03:   ...
04:   int b7:1;}
05: union U{
06:   struct ST s;
07:   char c;};
08: void f(char c){
09:   union U u;
10:   u.c = c;
11:   if(u.s.b7 == 1){ // concretized
12:     /* not covered */...}}
13: void driver_f(){
14:   char param1;
15:   SYM_char(param1);
16:   f(param1);}
```

Fig. 3. An example showing that the branch at Line 12 is not covered due to the lack of symbolic bit-field support

One naive solution can be to transform all bit-fields into integer variables. However, this approach changes the semantics of a target program when an integer overflow occurs to a bit-field or `union` is used with bit-fields. In Fig. 3, `union U` has two fields, `struct ST s` and `char c` which are located in the same memory space. Since `u.s` and `u.c` share the same memory space, the assignment of a value to `u.c` (Line 10) also updates all bit-fields `b0`, ..., `b7` in `u.s` at the same time. [5] Suppose that we transform the bit-fields in `struct ST` into integer variables. Then, the assignment of a (symbolic) value to `u.c` (Line 10) does not update `u.s.b7` because `u.s.b7` is not located in the same memory space of `u.c` anymore. Consequently, concolic testing may fail to reach Line 12 because `u.s.b7` is not symbolic.

3) No support for symbolic function pointers: IBU uses function pointers to make compact code. Current concolic testing tools, however, do not support symbolic function pointers due to the limitation of SMT solvers and fail to cover branches whose conditions depend on a function pointed by a function pointer p_f (i.e., concolic testing fails to generate various execution scenarios enforced by assigning different functions to p_f).

III. Mobis Automated Testing Framework

A. Overview

Fig. 4 overviews MAIST, which takes C source code files as inputs. MAIST consists of the three components: *test harness generator*, *converter*, and *test input generator*. First MAIST harness generator analyzes the input C source files and generates test driver and stub functions for every task in the source files (Sect. III-C). MAIST converter transforms the C code that uses bit-fields into semantically equivalent one that does not use bit-fields (Sec. III-D). Finally, MAIST test input generator performs concolic testing using CROWN (Concolic

[3] We exclude Linux Kernel because our Clang-based analysis tool fails to analyze GCC-specific code of Linux Kernel.

[4] A concolic testing tool maintains a symbolic memory which maps a *memory address* to a corresponding symbolic variable. A concolic testing tool cannot get a memory address of a bit-field because the address-of operator (i.e., `&`) in C does not take a bit-field as an operand.

[5] IBU often uses this code pattern to update multiple bit-fields at once.

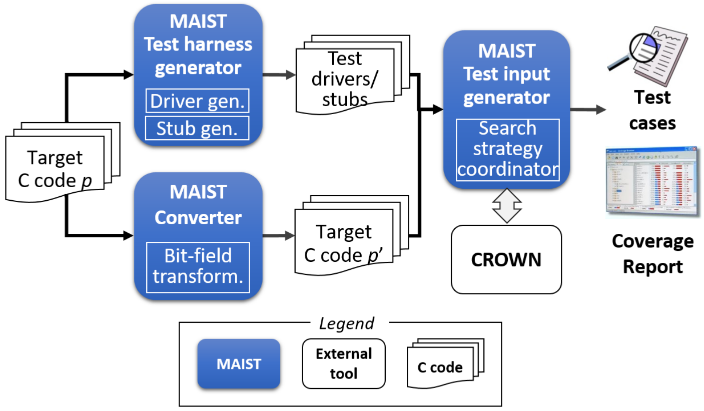


Fig. 4. The overview of MAIST

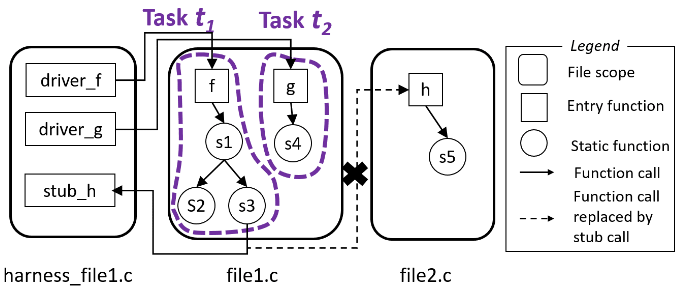


Fig. 5. Test driver and stubs generated for tasks t_1 and t_2 in `file1.c`

testing for Real-wOrld softWare aNalysis) [14] [6] to generate test inputs obtained by applying various symbolic search strategies (Sect. III-E).

B. MAIST Implementation

The development team of MAIST consists of two Mobis engineers (one senior and one junior engineer) and three researchers of KAIST. The team spent four months to implement MAIST. MAIST test harness generator is implemented in 3,500 lines of code (LoC) in C++ using Clang/LLVM 4.0 [16]. MAIST converter is implemented in 1,100 LoC in OCaml using CIL (C Intermediate Language) 1.7.3 [17]. We chose CIL for MAIST bit-field transformer because the canonical C code generated by CIL makes implementation of the bit-field transformer easy. MAIST test input generator is implemented in 200 LoC in Bash shell script. MAIST test input generator uses CROWN [14] because CROWN (and its predecessor CREST) has been successfully applied to various industrial projects (Sect. VII-C) and two of the authors are involved in developing CROWN.

C. MAIST Test Harness Generator

1) Task-Oriented Driver and Stub: Fig. 5 shows an example of generating test drivers and stubs for two tasks t_1 and t_2. Suppose that a target program p consists of `file1.c` and `file2.c`. `file1.c` has two non-static functions `f` and `g` and four static functions `s1` to `s4`. t_1 consists of the entry function `f` and its callee functions `s1`, `s2` and `s3` defined in the same file (i.e., `file1.c`). The function `h` invoked by `s3` is not included in t_1 because `h` is defined in another source file (i.e., `file2.c`). Similarly, t_2 consists of the entry function `g` and its callee function `s4`. MAIST targets a task t as a testing target unit and generates a test driver to invoke a task entry function t_e and test stubs to replace the callee functions of the task t located in other source files.

2) Automated Generation of Test Drivers and Stubs: MAIST automatically generates a test driver that invokes the entry function t_e of each task. The test driver assigns symbolic values to the arguments of t_e and to the global variables used in the task t for concolic testing and invokes t_e. Also, MAIST generates test stubs for the functions that are not included in the task (e.g., `h` for t_1) which return unconstrained symbolic values.

In Fig. 5, MAIST generates `harness_file1.c` that contains test drivers and a stub function for t_1 and t_2 in `file1.c`. `driver_f` and `driver_g` are test driver functions that call the entry functions `f` and `g` of t_1 and t_2, respectively. `s3` in t_1 invokes `h` which is not defined in `file1.c`. Thus, MAIST generates a stub function `stub_h` in `harness_file1.c` and modifies `s3` to call `stub_h` instead of `h`.

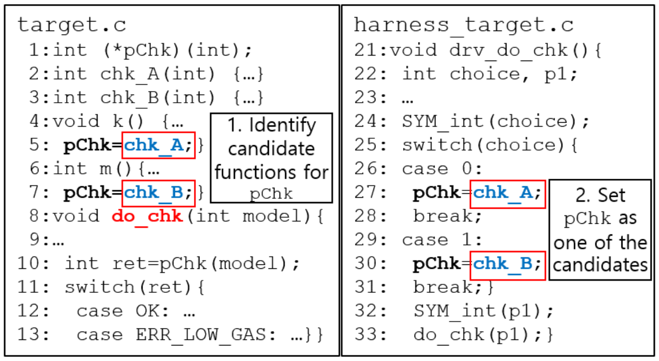
MAIST specifies a variable as a symbolic input according to its type as follows:

- *Primitive types:* MAIST specifies a primitive variable `x` as a symbolic input by using `SYM_<T>(x)` where `<T>` is a type of `x`.
- *Array types:* MAIST specifies each array element as a symbolic input according to the type of the element.
- *Pointer types:* For a pointer `p` pointing to a memory of a type `T`, MAIST allocates memory in `sizeof(T)*n` bytes where `n` is a user-given bound (i.e., MAIST considers `p` to point to an array which has `n` elements and whose element type is `T`).
- *Structure types:* For a `struct` variable `s`, MAIST specifies each field of `s` as a symbolic input according to the field type recursively. To prevent infinite recursive dereference (e.g., a linked list forming a cycle), MAIST follows a pointer to `s` within a user-given bound k and assigns `NULL` to a pointer that is not reachable within the bound.
- *Bit-field types:* MAIST specifies a bit-field `b` as a symbolic input by using `SYM_bitfield(b)`.

3) Symbolic Setting for Function Pointers: MAIST generates a test driver which assigns various functions to a function pointer p_f used in a target task. First, MAIST statically examines all target source files and identifies functions $f_1, ...f_n$ that are assigned to p_f. Then, it adds code c_{p_f} to a test driver such that c_{p_f} assigns each of $f_1, ..., f_n$ to p_f using a symbolic variable *choice* which selects each of $f_1, ..., f_n$ to assign to p_f.

For example, Fig. 6 has `do_chk` (Lines 8–13) which has branches whose conditions depend on `ret` (Lines 11–13). `ret` value is determined by the return value of the function pointed to by `pChk` (Line 10). After MAIST identifies that

[6] CROWN: Concolic testing for real-world software analysis, http://github.com/swtv-kaist/CROWN, accessed: 2018-10-01.



```
target.c
 1:int (*pChk)(int);
 2:int chk_A(int) {…}
 3:int chk_B(int) {…}
 4:void k() {…
 5: pChk=chk_A;}
 6:int m(){…
 7: pChk=chk_B;}
 8:void do_chk(int model){
 9:…
10: int ret=pChk(model);
11: switch(ret){
12:   case OK: …
13:   case ERR_LOW_GAS: …}}
```

```
harness_target.c
21:void drv_do_chk(){
22: int choice, p1;
23: …
24: SYM_int(choice);
25: switch(choice){
26: case 0:
27:   pChk=chk_A;
28:   break;
29: case 1:
30:   pChk=chk_B;
31:   break;}
32:   SYM_int(p1);
33:   do_chk(p1);}
```

1. Identify candidate functions for pChk
2. Set pChk as one of the candidates

Fig. 6. Example of symbolic setting for function pointer pChk

Items	Original program *p*	Transformed program *p'*

struct definition	struct S1{
	   int a:2;
	   int b:3;
	   int c:3;
	} s1;
		struct S2{
		  unsigned char bits[1];
		} s2;

Bit-field read	if (s1.b == 5)	if((s2.bits[0] & 0b00011100)>>2)==5)

Bit-field write	s1.b = 6;	changed[0] = (6 << 2) & 0b00011100;
		unchanged[0]= s2.bits[0] & 0b11100011;
		s2.bits[0]= changed[0] | unchanged[0]

Fig. 7. An example of bit-field transformation

pChk may point to chk_A or chk_B at Line 5 or Line 7 respectively, it adds code c_{p_f} to a test driver drv_do_chk which assigns chk_A and chk_B to pChk depending on a symbolic variable choice (Lines 25–31), as shown in the right part of Fig. 6.

4) Test Driver Generation for a Task with Internal States: Some IBU functions use static local variables to keep the execution results of the previous invocation as its internal state. For a task containing such a function, MAIST generates a test driver to call a target task multiple times with fresh symbolic inputs.

D. MAIST Converter

Automotive software uses *bit-fields* to minimize the data size. Unlike other primitive types in C which occupy multiples of 8 bits, the size of a bit-field does not have to be in multiples of 8 bits and can be smaller than 8. Currently, concolic testing tools for C programs such as CREST [18], CUTE [5], KLEE [19], and PathCrawler [20] do not support symbolic declaration of bit-fields and fail to achieve high coverage of automotive software that uses bit-fields (Sect. II-D2).

To solve this problem, MAIST transforms a target program p into p' that is semantically equivalent to p but does not use bit-fields. In other words, MAIST replaces bit-fields with a data array of a byte type and also replaces all arithmetic expressions on the bit-fields with semantically equivalent ones without the bit-fields using the data array with bit-wise operators as follows:

- struct *definition:* For a struct definition S1 which has bit-fields, MAIST transforms S1 to struct S2 whose size is same to S1. All fields in S1 including bit-fields are represented by a data array unsigned char bits[sz] in S2 (sz is a byte size of S1) as follows. A field b in S1 whose bit-offset is m bits and size is s bits is mapped to a sequence of bits from $(m\%8)^{th}$ bit of S2.bits[$m/8$] to $((m+s-1)\%8)^{th}$ bit of S2.bits[$(m+s-1)/8$] where % is the modulo operator.
  For example, the middle column of Fig. 7 shows an original target program p that has bit-fields a:2, b:3, and c:3 in s1 (e.g., a bit-field b in s1 has a bit-offset $m = 2$ and a bit-size $s = 3$). The rightmost column of Fig. 7 shows a transformed version p' which does not have bit-fields but a data array bits representing all fields of s1.
- *Bit-field read:* MAIST transforms an expression on a bit-field b into an equivalent one using the data array bits, b's bit-offset m and size s, and bit-wise operators. For example, a value of b can be obtained by a transformed expression ((s2.bits[0] & 0b00011100)>>2).
- *Bit-field write:* MAIST transforms a bit-field write statement on a bit-field b into an equivalent one using s2.bits, b's bit-offset m and size s, and bit-wise operators as follows.
  First, MAIST obtains the sequence of the updated bits that ranges from $(m\%8)^{th}$ bit of s2.bits[$m/8$] to $((m+s-1)\%8)^{th}$ bit of s2.bits[$(m+n-1)/8$]. Then, it stores those bits into a temporary byte array changed. Second, MAIST obtains the two sequences of the remaining unchanged bits - one ranging from 0^{th} to $(m\%8-1)^{th}$ bits of s2.bits[m/8] and another ranging from $((m+n-1)\%8+1)^{th}$ bit of s2.bits[$(m+n-1)/8$] to the last bits of s2.bits. Then, it stores those bits into another temporary byte array, unchanged. Finally, MAIST updates the data array bits by merging changed and unchanged arrays using the bitwise-or operator (i.e., for all i, bits[i]=changed[i] | unchanged[i]).
  For example, in Fig. 7, where p has s1.b = 6, (6 << 2) & 0b00011100 is assigned to changed[0] in p'. Similarly, s2.bits[0] & 0b11100011 is assigned to unchanged[0] in p'. Then, s2.bits[0] is updated as changed[0] | unchanged[0].

E. MAIST Input Generator

MAIST input generator utilizes various symbolic search strategies to increase test coverage. Although there are dozens of symbolic search strategies [21] to increase coverage within a given time budget, no single strategy outperforms all others because they are heuristics by their nature.

MAIST search strategy coordinator utilizes the four search strategies (i.e., depth-first-search (DFS), reverse-DFS, random negation, and control-flow-graph based one (CFG)) to increase test coverage and reduce execution time. MAIST search strategy coordinator applies DFS as the first search strategy to explore all possible paths. This is because DFS stops concolic testing when it has explored all possible execution paths. If DFS has explored all possible execution paths, MAIST stops concolic testing for the target task. Otherwise, the search strategy coordinator applies reverse-DFS and random negation. Lastly, MAIST search strategy coordinator applies CFG to the remaining uncovered branches where CFG tries to guide concolic testing to reach the uncovered branches [22].

IV. Industrial Case Study: Application of MAIST to Integrated Body Unit

We have developed and applied MAIST to IBU from October 2017 to July 2018 as explained in the following subsections.

A. Research Questions

RQ1 to RQ3 evaluate the experiment results of applying MAIST to IBU.

RQ1. Effectiveness of the automated test generation: How much test coverage does MAIST achieve for IBU in terms of branch and MC/DC coverage?

RQ2. Analysis of the uncovered branches: What are the major reasons for MAIST to fail to reach uncovered branches?

RQ3. Benefit of MAIST over the manual testing: How much human effort does MAIST reduce in terms of the test engineer man-month spent for IBU?

RQ4 to RQ6 evaluate how effectively MAIST addresses the technical challenges described in Sect. II-D.

RQ4. Effect of the task-oriented automated test generation: Compared to a function-oriented technique, how much test coverage does MAIST achieve for IBU in terms of branch and MC/DC coverage and how many false crash alarms does MAIST raise?

RQ5. Effect of the symbolic bit-field support: How much does the symbolic bit-field support of MAIST increase the branch and MC/DC coverage?

RQ6. Effect of the symbolic setting for function pointer: How much does the symbolic setting provided by MAIST increase the branch and MC/DC coverage?

B. Test Generation Techniques Used

To evaluate the strengths and weaknesses of the test generation ability of MAIST, we have compared MAIST with the following test generation techniques:

- *MAIST using function-oriented concolic unit testing (MAIST^{FO})*: This variant of MAIST is the same as MAIST but generates test drivers and stubs in a function-oriented manner. MAIST^{FO} generates a test driver for each target function and replaces all the functions invoked by the target function with the stub functions. We compare MAIST to MAIST^{FO} for RQ4.
- *MAIST without symbolic bit-field support (MAIST^{-SBF})*: This variant of MAIST is the same as MAIST but does not support symbolic bit-fields (Sect. III-D). We compare MAIST to MAIST^{-SBF} for RQ5.
- *MAIST without symbolic setting for function pointers (MAIST^{-SFP})*: This variant of MAIST is the same as MAIST but the generated test driver by MAIST^{-SFP} does not provide symbolic setting for a function pointer (Sect. III-C3). We compare MAIST to MAIST^{-SFP} for RQ6.

C. Measurement

To show the test effectiveness of MAIST, we measure branch coverage and MC/DC coverage by using CTC++ [23]. We measure branch coverage because the manual test generation of IBU targets 100% of branch coverage in Mobis and we need to compare the manual test generation and MAIST for RQ3. Also, we measure MC/DC coverage because MC/DC coverage is required for safety critical components by ISO 26262 safety requirement for automotive systems. Also, to compare the number of false crash alarms generated by MAIST and MAIST^{FO}, we measure the number of crash alarms and crash locations by counting the number of test executions that cause a crash (e.g., segmentation fault) and the code lines where a crash occurs, respectively.

D. Test Configuration

For each target task, we set MAIST, MAIST^{FO}, MAIST^{-SBF}, and MAIST^{-SFP} to run until they each satisfy one of the following conditions:

1) All possible execution paths are explored, or
2) The automated test generation technique reaches 20 minutes timeout.

Since MAIST applies the four search strategies (Sect. III-E), each search strategy has five minutes as a timeout. For MAIST^{FO}, we set the timeout as four minutes (=(19.4 hours×60 minutes/hour×4 cores×3 machines)/3479 functions) for each target function to make the total amounts of testing time of MAIST^{FO} and MAIST same.

For the tasks that have a function with `static` local variables, MAIST generates a test driver that invokes the target task twice with fresh symbolic inputs. We chose the number of the repeated invocations as two because most tasks use `static` local variables to keep the immediately previous execution results. We set the user-given size bound n for pointers as 10 and k for `struct` variables as 4 (Sect. III-C2).

The experiments were performed on three machines, each of which is equipped with Intel Xeon X5670 (6-cores 2.93 GHz) and 8GB RAM, running 64 bit Ubuntu 16.04. We run four test generation instances on each machine (i.e., applying MAIST to 12 tasks (=4 instances×3 machines) in parallel).

V. Experiment Results

A. RQ1. Effectiveness of the Automated Test Generation

Table II shows a number of the generated test inputs, execution time, and branch and MC/DC coverage of IBU achieved by MAIST. MAIST generated 914,023 test inputs in 19.4 hours on three machines (i.e., on 12 cores), which achieved 90.5% branch coverage and 77.8% MC/DC coverage of IBU.

TABLE II
THE NUMBER OF GENERATED TESTS, EXECUTION TIME, AND BRANCH AND MC/DC COVERAGE OF IBU ACHIEVED BY MAIST

Targets	#tests	Time (hour)	Branch cov. (%)	# of func. achieving given branch cov. range						MC/DC cov. (%)	# of func. achieving given MC/DC cov. range					
				[0%, 20%)	[20%, 40%)	[40%, 60%)	[60%, 80%)	[80%, 100%)	100%		[0%, 20%)	[20%, 40%)	[40%, 60%)	[60%, 80%)	[80%, 100%)	100%
BCM	77981	3.7	91.4	0	14	52	65	86	439	78.9	0	38	50	122	143	303
SMK	761743	13.9	90.1	0	83	132	242	196	1868	77.3	0	98	246	276	347	1554
TPMS	74299	1.8	91.6	0	11	14	15	56	206	79.5	0	15	17	58	81	131
Total	914023	19.4	90.5	0	108	198	322	338	2513	77.8	0	151	313	456	571	1988

TABLE III
THE NUMBER AND RATIO OF THE UNCOVERED BRANCHES OF THE LARGE BCM FUNCTIONS (IN TERMS OF THE NUMBER OF BRANCHES) WHOSE BRANCH COVERAGE IS LESS THAN 60%

Reasons	#Uncovered branches	Ratio (%)
Unreachable branches	8	7.8
Path explosion	21	20.4
Imprecise driver	10	9.7
Imprecise stub	43	41.7
`static` local variable	21	20.4
Total	103	100

MAIST achieved 100% branch coverage and 100% MC/DC coverage of 72.2% (=2513/3479) and 57.1% (=1988/3479) of all functions in IBU, respectively. Also, MAIST achieved more than 80% branch and 80% MC/DC coverage for 81.9% (=(338+2513)/3479) and 73.6% (=(571+1988)/3479) of all functions in IBU, respectively.

B. RQ2. Analysis of the Uncovered Branches

We manually analyzed the uncovered branches of BCM as an example. Among the 66 (=0+14+52) functions in BCM whose branch coverage is less than 60%, we selected the largest 33 functions in terms of the number of branches. Table III shows the five reasons why MAIST did not cover the 103 uncovered branches of these 33 functions as follows:

- *Unreachable branches:* 8 branches (=7.8% (=8/103)) are unreachable code because the BCM version used in this experiment targets a specific motor vehicle model and these uncovered branches are designed to execute only for another motor vehicle model.
- *Path explosion:* 21 branches (=20.4%) are uncovered due to the path explosion problem of concolic testing. These branches can be covered if we increase the time limit for test generation per task (i.e., larger than 20 minutes).
- *Imprecise driver:* 10 branches (=9.7%) are uncovered because test drivers generated by MAIST provide only limited symbolic inputs for complex data structure (i.e., a test driver provides symbolic inputs for only variables reachable from a target task within a given pointer link bound (i.e., $k = 4$) (Sect. III-C2)). To cover these branches, MAIST has to increase the pointer link bound (e.g., $k > 4$). However, an increased pointer link bound may not increase the coverage within fixed testing time limit due to enlarged symbolic path space.
- *Imprecise stub:* 43 branches (=41.7%) are uncovered because the stub functions generated by MAIST do not symbolically set variables pointed by pointer parameters and global variables which are read by a target task. We will discuss this issue further in Sect. VI-C.
- `static` *local variable:* 21 branches (=20.4%) are uncovered because MAIST fails to assign diverse values to `static` local variables through symbolic input variables of a target task (Sect. III-C4).

TABLE IV
BRANCH AND MC/DC COVERAGE ACHIEVED AND CRASH LOCATIONS REPORTED BY MAIST^{FO} AND MAIST

Module	Branch coverage (%)		MC/DC coverage (%)		#crash alarms (and # crash lines)	
	MAIST^{FO}	MAIST	MAIST^{FO}	MAIST	MAIST^{FO}	MAIST
BCM	95.2	91.4	89.3	78.9	3479 (42)	0 (0)
SMK	95.3	90.1	88.9	77.3	6453 (79)	0 (0)
TPMS	95.1	91.6	87.9	79.5	1992 (25)	0 (0)
Total	95.3	90.5	88.9	77.8	11924 (146)	0 (0)

C. RQ3. Benefit of MAIST over the Manual Testing

We show the time cost of the manual testing of IBU and how much cost MAIST has reduced.

1) Cost of the Manual Testing: Previously, a team of 30 test engineers at Mobis had written test inputs for coverage testing of IBU. The test engineers have three years of experience in testing and QA on average. A test engineer writes function test inputs targeting 100% branch coverage for 350 LoC in one business day, on average (i.e., for one month, a test engineer writes unit test inputs for 7 KLoC (=350 LoC× 20 business days) on average). Thus, writing manual test inputs for coverage testing of IBU (210K LoC) requires 30 man-months (MM) ($= \frac{210KLoC}{7KLoC\ per\ month}$).

2) Benefit of MAIST: MAIST reduced 53.3% of the manual testing effort as follows. After applying MAIST, the test engineers still have to generate test inputs to cover 9.5% (= 100-90.5) of the IBU branches that were not covered by MAIST. The test engineers spent five MM to cover those branches. Also, nine MM were spent to develop MAIST and train the test engineers to use MAIST. Thus, 30 MM of the manual testing effort for coverage testing of IBU is reduced to 14 MM, which is equivalent to 53.3% of the previous manual coverage testing cost of IBU. Note that MAIST will reduce the manual testing effort much further for the future application since the cost of the nine MM for the development and training of MAIST is just one time cost.

TABLE V
BRANCH AND MC/DC COVERAGE ACHIEVED BY MAIST^{-SBF} AND MAIST

Target	Branch coverage (%)		MC/DC coverage (%)	
	MAIST^{-SBF}	MAIST	MAIST^{-SBF}	MAIST
129 tasks using bit-fields	48.3	89.4	38.2	83.6

TABLE VI
BRANCH AND MC/DC COVERAGE ACHIEVED BY MAIST^{-SFP} AND MAIST

Target	Branch coverage (%)		MC/DC coverage (%)	
	MAIST^{-SFP}	MAIST	MAIST^{-SFP}	MAIST
88 tasks using function pointers	68.2	91.3	53.0	80.5

D. RQ4. Effect of the Task-oriented Automated Test Generation

We compare the branch and MC/DC coverage and the number of the crashes reported by MAIST^{FO} and MAIST. Table IV shows that MAIST^{FO} achieves 5.3% (= $(95.3 - 90.5)/90.5$) and 14.3% higher branch and MC/DC coverage than MAIST, respectively. This is because MAIST^{FO} directly controls the executions of each function f by generating test inputs to f while MAIST controls f indirectly through the entry function of the task that contains f.

However, MAIST^{FO} generated many infeasible test inputs and raised 11,924 false crash alarms. This is because MAIST^{FO} directly generates inputs for every function f that violate the context of f provided by the caller and callee functions of f. We semi-automatically analyzed all 11,924 crash alarms at 146 lines in IBU reported by MAIST^{FO} and found that all reported crash alarms were false. [7] In contrast, MAIST did not raise any crash alarm because it provides valid test inputs to f indirectly through the entry function of the task of f. Thus, we can conclude that MAIST reports more reliable coverage information than MAIST^{FO}.

E. RQ5. Effect of the Symbolic Bit-field Support

For the 129 tasks that use bit-fields, we compare the branch and MC/DC coverage achieved by MAIST^{-SBF} and MAIST. Table V shows that MAIST achieved 1.9 times higher branch coverage and 2.2 times higher MC/DC coverage than MAIST^{-SBF}. Since MAIST^{-SBF} does not generate test inputs for bit-fields at all, it may not cover the branches whose conditions depend on bit-fields (Sect. II-D2). Thus, we can conclude that this support of symbolic bit-fields increases test coverage for automotive software such as IBU.

F. RQ6. Effect of the Symbolic Setting for Function Pointers

For the 88 tasks that use function pointers, we compare the branch and MC/DC coverage achieved by MAIST^{FO} and MAIST. Table VI shows that MAIST achieved 33.9% and 51.9% higher branch and MC/DC coverage than MAIST^{-SFP}, respectively. Since MAIST^{-SFP} does not set function pointers, the branches that have control-dependency on the function invoked through a function pointer may not be covered by MAIST^{-SFP}. Thus, we can conclude that symbolic setting for function pointers increases test coverage for automotive software.

VI. LESSONS LEARNED

A. Practical Benefit of Automated Test Generation in the Automotive Industry

As we have seen in Sect. V-A and V-C, an automated test generation technique such as MAIST can improve the quality of automotive software by both achieving high test coverage (i.e., more than 90% branch coverage) and saving the testing cost (i.e., 53.3% man-month per year on coverage testing of IBU) in practice. Although it is not trivial to develop an automated test generation framework that resolves various technical challenges in industrial projects, we believe that the automotive industry can significantly benefit from an automated test generation framework like MAIST.

B. Necessity of Customization of Automated Test Generation Tools for Target Projects

From this industrial study on the automotive software, we have found that it is essential to identify technical challenges and customize an automated test generation tool to address those challenges in a target project. For example, if MAIST targeted an individual function as a target unit (not a task), it would generate misleading coverage information and waste human effort to filter out false alarms due to infeasible test inputs generated (Sect. V-D), which would reduce the benefits of MAIST. Or, the proposed task-oriented approach might not be highly effective for other projects. Also, if MAIST did not support bit-fields nor symbolic setting for function pointers, MAIST would not achieve 90% branch coverage, but much less coverage.

C. Precise Stub Generation for Automated Test Generation

As shown in Sect. V-B (i.e., 41.7% of the uncovered branches were due to the imprecise stubs), we need to generate precise stubs that closely represent real contexts of a target unit. A stub generated by MAIST does not represent a target task's context accurately because the stub function sets only a return value (not global variables nor output parameter variables) as an unconstrained symbolic input.

This issue is difficult to resolve because a simple solution such as assigning unconstrained symbolic inputs to all global variables and output pointer parameters used by a target task may generate more infeasible test executions and increase symbolic execution space, which may decrease test effectiveness within a fixed testing time. Although there has been progress in resolving this issue (Sect. VII-B), we still need practical techniques to generate more precise stubs that closely mimic the real context of a target unit.

[7] First, we classified the 11,924 crashing test inputs in 210 groups by filtering out the input values irrelevant to the crashes at the 146 crash lines. Then, we manually analyzed 210 test inputs, each of which represents a group of the crashing test inputs.

VII. Related Work

A. Concolic Testing Techniques

1) Virtual machine (VM)-based Techniques: These techniques run as a layer on top of a VM to interpret compiled IR code of a target program p and obtain symbolic path formulas from p's executions. This approach can conveniently obtain all detailed run-time execution information of p available to a VM. However, test generation speed is slow due to slow IR interpretation and customizing the tools is non-trivial due to complex VM infrastructure. PEX [6] targets C# programs that are compiled to Microsoft .Net binaries (now available as IntelliTest [24] in Visual Studio). KLEE [19] (and its distributed version Cloud9 [25]) targets LLVM [16] binaries. jFuzz [26] and Symbolic PathFinder [27] target Java bytecode programs on top of Java PathFinder [28].

2) Instrumentation-based Techniques: These techniques insert probes in target source code to obtain dynamic execution information to build symbolic path formulas. This approach is lighter and easier-to-customize than the VM-based one. However, it requires complex source code parsing and instrumentation. CUTE [5], DART [29], CREST [22] (and its distributed version SCORE [30]), CROWN [14] target C programs and jCUTE [31] and CATG [32] target Java programs. MAIST uses CROWN as its concolic testing engine because CROWN (and its predecessor CREST) has been successfully applied to various industrial projects (Sect. VII-C).

B. Automated Test Driver/Stub Generation

DART [29] generates symbolic unit test drivers, but not symbolic stubs for concolic testing. To avoid the infeasible test generation issue, DART targets public API functions in libraries because such functions should accept all possible inputs. UC-KLEE [33] directly starts symbolic execution from a target function using lazy initialization [34] and calls all the functions directly or transitively invoked by the target function. Thus, DART and UC-KLEE target code of a function and its all callee functions, which can make concolic testing achieve low coverage within a fixed amount of testing time because the symbolic execution space can become very large. Chakrabarti and Godefroid [35] statically divide a static call graph into partitions using topological information and consider the partitions as testing targets for concolic testing. However, the proposed partitioning method does not consider semantic information on the relation between functions.

CONBOL [10], [36] generates symbolic unit test driver and stubs for functions in large-scale embedded software. It replaces all functions invoked by a target function by symbolic stubs. CONBOL uses target project specific false alarm reduction heuristics, which may not be effective for other projects. SmartUnit [37] generates symbolic test drivers and stubs for target C functions (the authors do not clearly describe how SmartUnit generates driver and stubs). The paper reports that SmartUnit achieved high coverage, but it does not report how many false alarms were raised. We could not directly compare the performance of MAIST with CONBOL and SmartUnit since they are not publicly available.

CONBRIO [14] constructs extended units as testing target units by using highly-relevant callee functions of a target function. The relevance between functions is computed based on system-level execution profiles. Targeting the extended units, CONBRIO achieves high bug detection power with low false alarm ratio. However, CONBRIO was not applicable to IBU in this testing project because we could not obtain IBU execution profiles by driving physical motor vehicles.

C. Industrial Application of Concolic Testing

Microsoft developed SAGE [38], [39] for x86/64 binaries to detect security vulnerabilities of Windows and Office products. Bardin and Herrmann [40], [41] developed and applied OSMOSE to embedded software. They translated machine code into an intermediate representation to apply concolic testing. Intel developed and applied MicroFormal [42] for Intel CPU's microcode. Fujitsu developed KLOVER [43] by extending KLEE targeting C++ programs. Zhang et al. developed and applied SmartUnit [37] to embedded software to achieve high branch and MC/DC coverage. Kim et al. applied CREST to the Samsung flash memory device driver code [7], [44]. They also compared CREST and KLEE for industrial use of concolic testing for the Samsung mobile phone software [9] and developed a systematic event-sequence generation framework using CREST for LG electric oven [45].

These industrial case studies focused on increasing the test effectiveness but did not report how much manual testing effort was saved. In contrast, this paper reports how much MAIST saved the manual testing effort (in man-months) in the automotive company (Sect. V-B). Also, we have shared the technical challenges and the solutions for the application of concolic testing to automotive software, which can promote field engineers to adopt concolic testing in their projects.

VIII. Conclusion

We have presented the industrial study of applying concolic testing to the automotive software developed by Mobis. After we identified and addressed the technical challenges of applying concolic testing to automotive software, we have developed an automated test generation framework MAIST. It generates a task-oriented test driver and stubs to reduce infeasible test executions and supports symbolic bit-fields and symbolic setting for function pointers that automotive software uses. MAIST has achieved 90.5% branch coverage and 77.8% MC/DC coverage on IBU and reduced the manual coverage testing effort by 53%. As future work, we plan to generate more precise test driver/stubs to reach uncovered branches. Also, we will apply MAIST to other automotive software to increase the economic benefit of MAIST.

Acknowledgement

We would like to thank Acheong Lee and Hyunwoo Kim for their initial effort of applying MAIST to IBU. This research has been supported by Hyundai Mobis, Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (No. NRF-2017M3C4A7068177), Basic Science Research Program through NRF funded by MSIT (NRF-2016R1A2B4008113), Basic Science Research Program through NRF funded by the Ministry of Education (NRF-2017R1D1A1B03035851), and MSIT, under the National Program for Excellence in SW (2016-0-00018), supervised by the IITP.

References

[1] M. Broy, "Challenges in automotive software engineering," in *International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2006, pp. 33–42.

[2] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356–373, Feb 2007.

[3] F. Falcini, G. Lami, and A. M. Costanza, "Deep learning in automotive software," *IEEE Software*, vol. 34, no. 3, pp. 56–63, May 2017.

[4] M. Traub, A. Maier, and K. L. Barbehön, "Future automotive architecture and the impact of it trends," *IEEE Software*, vol. 34, no. 3, pp. 27–32, May 2017.

[5] 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, pp. 263–272.

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

[7] M. Kim, Y. Kim, and Y. Choi, "Concolic testing of the multi-sector read operation for flash storage platform software," *Formal Aspects of Computing*, vol. 24, no. 3, pp. 355–374, May 2012.

[8] M. Kim, Y. Kim, and Y. Jang, "Industrial application of concolic testing on embedded software: Case studies," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 390–399.

[9] 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, pp. 1143–1152.

[10] 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 *Automated Software Engineering (ASE)*, 2013, pp. 519–528.

[11] "Hyundai mobis increases ai use for improved efficiency," http://www.koreaherald.com/view.php?ud=20180722000186, accessed: 2018-10-10.

[12] "Hyundai mobis taps ai for car software," https://www.koreatimes.co.kr/www/tech/2018/07/419_252629.html, accessed: 2018-10-01.

[13] "Hyundai mobis introduces ai-based software verification system," http://www.businesskorea.co.kr/news/articleView.html?idxno=23830, accessed: 2018-10-01.

[14] Y. Kim, Y. Choi, and M. Kim, "Precise concolic unit testing of c programs using extended units and symbolic alarm filtering," in *International Conference on Software Engineering (ICSE)*, 2018, pp. 315–326.

[15] "Open hub, the open source network," https://www.openhub.net/.

[16] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04, 2004.

[17] 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.

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

[19] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, pp. 209–224.

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

[21] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, May 2018.

[22] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-123, Sep 2008.

[23] "Testwell CTC++," https://www.verifysoft.com/en_ctcpp.html, accessed: 2018-10-01.

[24] N. Tillmann and J. De Halleux, "Pex: White box test generation for .NET," in *Proceedings of the 2Nd International Conference on Tests and Proofs*, ser. TAP'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 134–153.

[25] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: a software testing service," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2010.

[26] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A concolic whitebox fuzzer for Java," in *NFM*, 2009.

[27] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008, pp. 15–26.

[28] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *ASE*, Sep. 2000.

[29] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[30] M. Kim, Y. Kim, and G. Rothermel, "A scalable distributed concolic testing approach: An empirical evaluation," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2012, pp. 340–349.

[31] K. Sen and G. Agha, "CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools," in *CAV*, 2006.

[32] "Catg: Concolic testing engine for java," https://github.com/ksen007/janala2, accessed: 2018-10-01.

[33] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15, 2015, pp. 49–64.

[34] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'03, 2003, pp. 553–568.

[35] A. Chakrabarti and P. Godefroid, "Software partitioning for effective automated unit testing," in *Proceedings of the 6th International Conference on Embedded Software*, ser. EMSOFT '06. New York, NY, USA: ACM, 2006, pp. 262–271.

[36] T. Kim, J. Park, I. Kulida, and Y. Jang, "Concolic testing framework for industrial embedded software," in *2014 21st Asia-Pacific Software Engineering Conference*, vol. 2, Dec 2014, pp. 7–10.

[37] C. Zhang, Y. Yan, H. Zhou, Y. Yao, K. Wu, T. Su, W. Miao, and G. Pu, "Smartunit: Empirical evaluations for automated unit testing of embedded software in industry," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 296–305.

[38] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Programming Language Design and Implementation (PLDI)*, 2008.

[39] "Microsoft security risk detection," https://www.microsoft.com/en-us/security-risk-detection/.

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

[41] ——, "OSMOSE: automatic structural testing of executables," *Journal of Software Testing, Verification, and Reliability (STVR)*, vol. 21, no. 1, pp. 29–54, 2011.

[42] A. Franzén, A. Cimatti, A. Nadel, R. Sebastiani, and J. Shalev, "Applying smt in symbolic execution of microcode," in *Formal Methods in Computer Aided Design*, Oct 2010, pp. 121–128.

[43] G. Li, I. Ghosh, and S. P. Rajan, "Klover: A symbolic execution and automatic test generation tool for c++ programs," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds., 2011, pp. 609–615.

[44] M. Kim and Y. Kim, "Concolic testing of the multi-sector read operation for flash memory file system," in *Brazilian Symposium on Formal Methods*, 2009.

[45] Y. Park, S. Hong, M. Kim, D. Lee, and J. Cho, "Systematic testing of reactive software with non-deterministic events: A case study on lg electric oven," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 29–38.